This course provides an introduction to RapidIO as it is applied on the PowerQUICC III family.

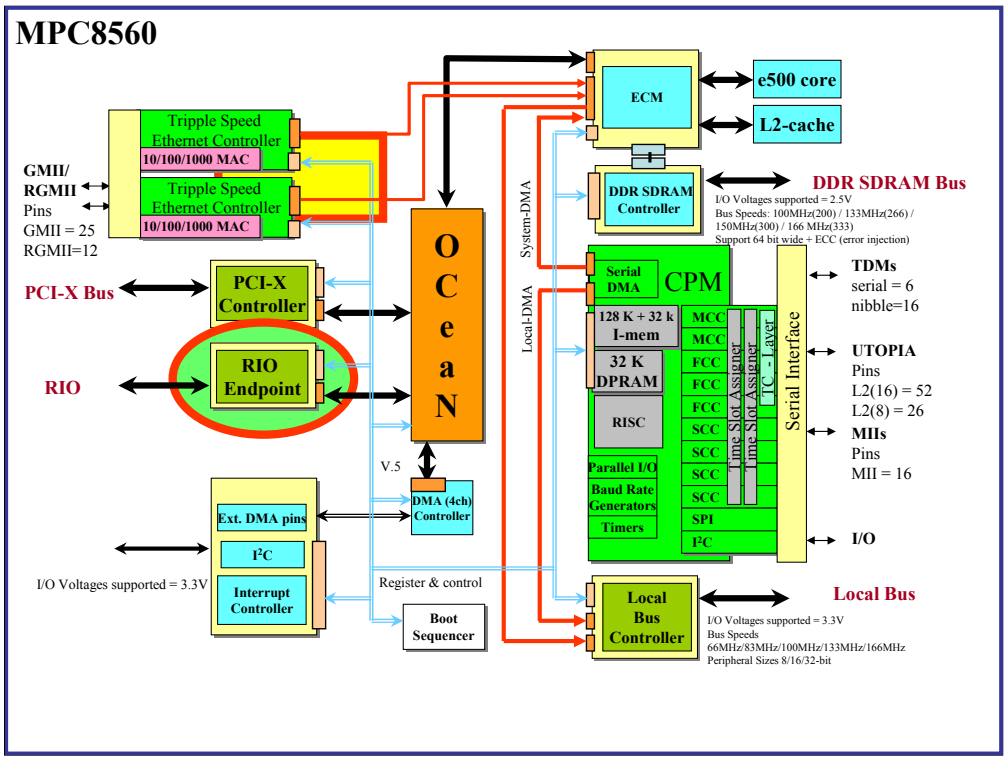
Introduction

This course provides an introduction into the operation of RapidIO and how the RapidIO controller operates on the PowerQUICC III family of processors.

The training covers a brief overview of the protocol operation,
Connections,
Control symbols,
Packet format,
Controller addressing and memory map,
Address translation and mapping,
How the message unit controller operates,
and an overview of the programming registers..

This course provides an introduction into the operation of RapidIO and how the RapidIO controller operates on the PowerQUICC III family of processors.

The training covers a brief overview of the protocol operation,
Connections,
Control symbols,
Packet format,
Controller addressing and memory map,
Address translation and mapping,
How the message unit controller operates,
and an overview of the programming registers.



This is a block diagram of the MPC8560 showing all the major blocks, and in particular the RapidIO.

A significant feature to notice here is the connections.

The RapidIO is connected into the 8560 via OCeaN, and OCeaN connects the PCI bus; DMA controller; and core block. For that reason data associated with those devices can be routed via the RapidIO .

Documentation

www.Rapidio.org

RapidIO Hardware Inter-operability Platform (HIP) document
Rev. 1.0

RapidIO Interconnect Specifiaction
Part V : Globally Shared Memory Logical Specification
Rev. 1.2
RapidIO Interconnect Specification Rev. 1.2
I/O Logical
Message Passing
Common Transport
Physical Layers

RapidIO Interconnect Specification
Part VII : System and Device Inter-operability Specification
Rev. 1.2

MPC8560 User's Manual section 16

This slide indicates the documents used in generating this course material.

The RapidIO web site has all the specifications for the different layers. It's important to be aware that each of those specifications deals with each individual layer as if it is independent, and so need to be taken in context. A good starting place for an overall view is the Interconnect specification.

Features supported by the PowerQUICC III

- 8- bit interface as defined by the 8/16 LP-EP physical layer Spec.
- non-coherent I/O transactions (not GSM)
- message passing
- CRC16 based error detection
- Up to 256 byte data payloads
- Packet pacing/retry capability (throttle control)
- Only small size transport information field (tt = 0b00)[256 device ID]
- Error recovery, training, time-out support
- RapidIO error injection

These are the RapidIO features supported by the PowerQUICC III.

An eight bit interface as defined by the physical layer specification

non-coherent input output transactions. Unless the data is transferred to or from the core then the data is not cached, and the controller does not support global shared memory.

message passing

Error detection using sixteen bit CRC checking

Message payloads can be up to a maximum of 256 bytes

Packet pacing and retry capability to control data transfer rates known as throttle control.

The PowerQUICC III only supports the small size transport information field. In other words the identification field is eight bits allowing I.D. numbers from 0 to 255.

Error recovery, training, and time out support

For testing there is an error injection function.

Features not supported by the PowerQUICC III

- Large size transport information (tt = 0b01)
- tod-sync control symbols (treated as idles) [ref multi-cast control symbol]
- ATOMIC swap and ATOMIC test-and-swap transactions [MEMC doesn't]
- Inbound buffer status is not reported (Max buffers is always reported)

I.D. numbers greater then eight bits are not supported.

tod-sync control symbols, which otherwise are used for multi-cast control, are not supported and are treated as idles.

Atomic swap and atomic test and swap transactions are not supported due to the memory controller not supporting them.

Inbound buffer status is not reported.

Modes of operation

- Transmit clock select mode - internal or receive clock [test only]
- CRC checking - enabled or disabled
- RapidIO transmission error checking - enabled or disabled
- RapidIO Port disable
- RapidIO Port power-down
- Accept all modes, disregard ID - “promiscuous”
- Packet Response time-out disable
- Link response time-out disable

The transmit clock input can be selected from an internal source or the receive clock. However, the receive clock should only be used for test mode because there is no PLL to ensure accurate tracking.

CRC can be disabled if required

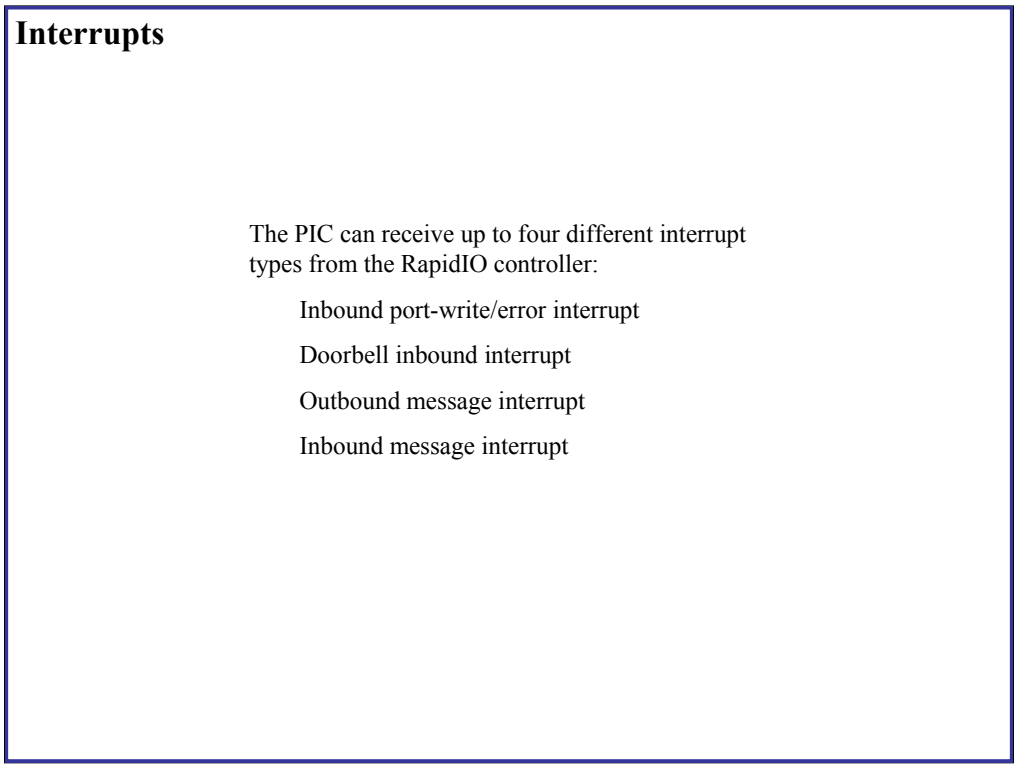
Transmission error checking can be disabled

There are two methods to disable the RapidIO function. The port can simply be disabled which leaves the controller operational, or it can be powered down when it is totally non-operational and not draining power.

Accept all mode, which means it disregards I.D. numbers and receives all incoming data.

The packet response timer can be disabled so that there is no time out mechanism for long (or non-existent) responses.

The link response time can also be disabled offering the same feature for maintenance symbols.


Interrupts
The PIC can receive up to four different interrupt types from the RapidIO controller:
Inbound port-write/error interrupt
Doorbell inbound interrupt
Outbound message interrupt
Inbound message interrupt

There are four possible interrupts generated by the RapidIO controller,

Inbound port-write and error conditions,
Doorbell inbound
Outbound message
and Inbound message

As will be seen subsequently, as with most of the functions of the PowerQUICC III, the interrupt service routine will have work to do to identify the exact nature of the service required.

Protocol Overview

- Protocol is made up of Packets and Control Symbols
  - Packets provide logical transaction interface between endpoints
    - Packets are transmitted end to end in the system
    - example - read, write, message
    - Packets contain a transport address (Source and Destination)
  - Control Symbols provide the physical layer control for transactions
    - Control Symbols are transmitted between two adjacent devices
    - example - acknowledge, retry, end of packet
    - Control Symbols contribute to hardware based error recovery
    - Control Symbols can be embedded within packets`

RapidIO is made up of packets and control symbols where the packets provide a logical transaction interface between endpoints, and the control symbols provide the physical layer control for transactions.

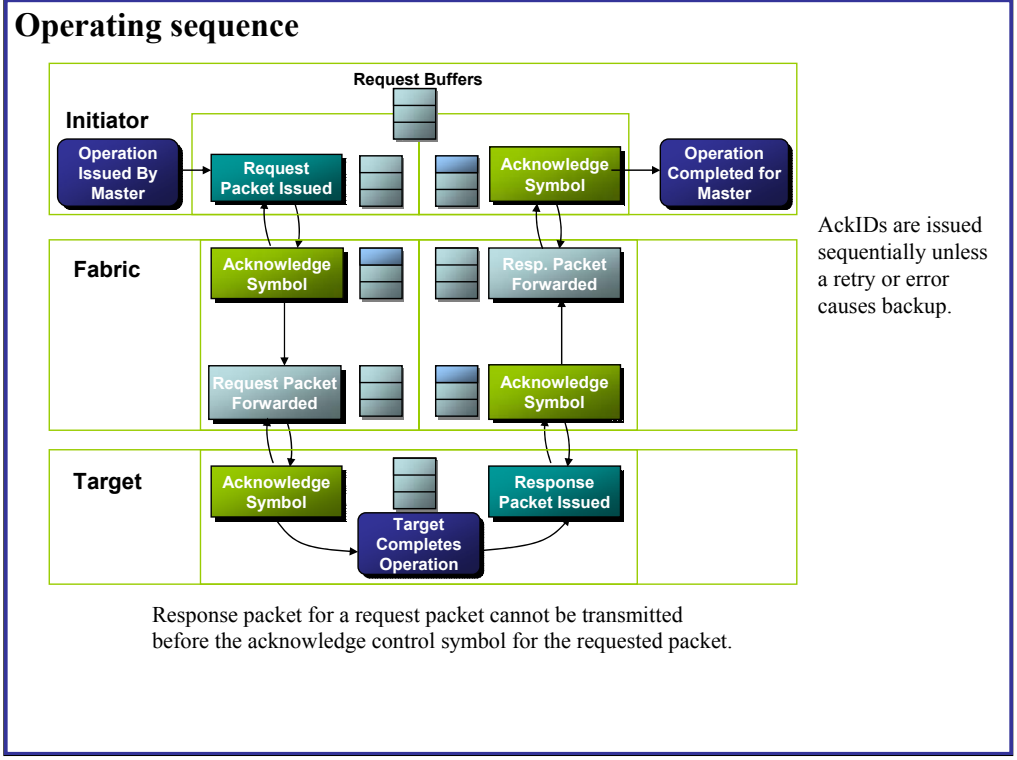
Packets are transmitted end to end, and so the protocol has a point to point connection, not a bus.

For example transactions can be a data read, write, or pass a message.

Each packet contains transport information including source and destination identification.

Control symbols are transferred between two adjacent devices providing such information as acknowledging a previous transaction, requesting a retry due to failure to process the transaction, or indicating the end of a packet.

They contribute to hardware based error recovery and can be embedded in message packets.



This animation will show how the sequence of events takes place during a transaction.

Here there are two endpoints with some switch fabric between. All transactions pass through the switch fabric, and there is a requirement for each transaction to be validated.

When an operation is issued by the master the controller provides two internal buffers to ensure that all transactions are dealt with before the operation is discarded.

A request packet is transferred to the switch fabric, so the switch will also allocate a buffer to this transaction to keep it until it has been safely passed on.

The switch acknowledges the transfer with an acknowledge symbol, and since the request has been dealt with the switch de-allocates the buffer, but also allocates another buffer for transferring the request onwards.

When the sender receives the acknowledgement indicating a successful transfer it's buffer is de-allocated, but the request buffer remains.

This process ensures that the sender knows that the request was transferred to the next stage, but at this stage the request has not been responded to by the target.

The switch fabric forwards the request to the target, and in the same manner the target allocates a buffer to the transaction.

The target responds with an acknowledge symbol to the switch, which now knows that the packet has been successfully transferred, and so de-allocates it's buffer, but the target keeps it's buffer until the transaction has been responded to.

The target deals with the request and a response packet is issued to the switch, which allocates a buffer to the transaction to ensure it is not lost for some reason.

This could have been a read operation where the initiator was requesting data, and so now the target has fetched that data and is now sending it to the initiator.

The switch responds with an acknowledge symbol to the target, so now both buffers are de-allocated, and the switch forwards the packet through the fabric allocating a buffer for the next step.

The response packet is transferred to the initiator which again allocates a buffer to the transaction, and after responding with the acknowledgement both buffers are de-allocated.

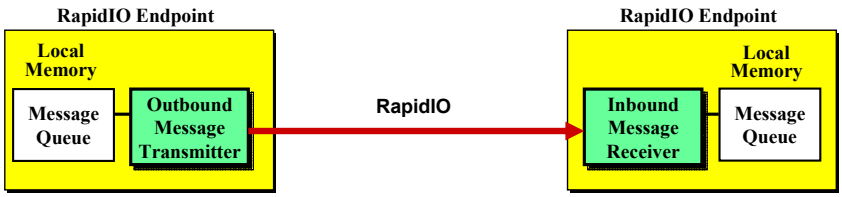
When the operation is complete the initiator de-allocates the request buffer. As you can see each step of the operation has the safety of the knowledge that a transfer has been completed, and if an acknowledgement is not received can recognize the problem and retry the transfer using the buffer allocated. Not only is each individual step protected, but also the initiator is aware if and when the operation is completed, and can handle errors if recognized. Each transfer is provided with an acknowledgement identification so that they are independently recognizable.

To ensure that there is no confusion, a response packet cannot be transmitted before the request packet has been acknowledged.

In reality there might not be a single operation from end to end as shown here, but a stream of requests and messages, and so the acknowledgement identification keeps track of which transactions have been dealt with, and each stage will allocate buffers for each transaction.

PowerQUICC III message passing

- Message and Doorbell Operations
  - Port based operations allowing devices to communicate with each other without direct visibility into each others address space
    - The receiver defines the memory location for the message to be buffered
- Doorbell
  - 16-bit message sent to target device
  - Useful for Interrupts
- Message
  - Inter-device communication through explicit messages instead of through direct memory access
  - A message can consist of up to 16 packets of up to 256 bytes (4k bytes max)
  - Supports all Mailboxes but only one.



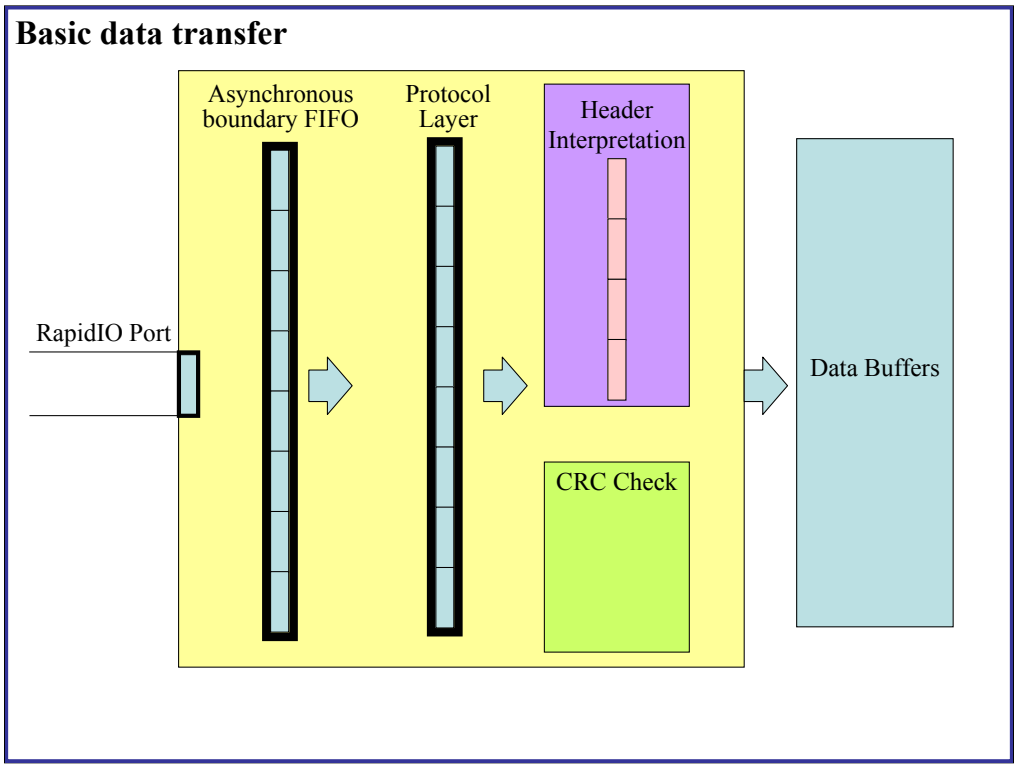
Messages and doorbells are port based operations that allow devices to communicate with each other without visibility to each other's address space.

The packet contains an identification value pertaining to the message, and the receiver defines the memory location for the message to be buffered.

A doorbell is a simple sixteen bit message sent to a target device, which is useful for generating an interrupt.

A message provides inter-device communication through explicit messages instead of direct memory access. It can consist of up to sixteen packets, where each packet can contain up to 256 bytes, enabling messages up to a maximum of four kilobytes.

The PowerQUICC III supports multiple mailboxes since any mailbox number can be received. However, because there is only one queue with no method of discriminating mailbox numbers there is no way this can be utilized.

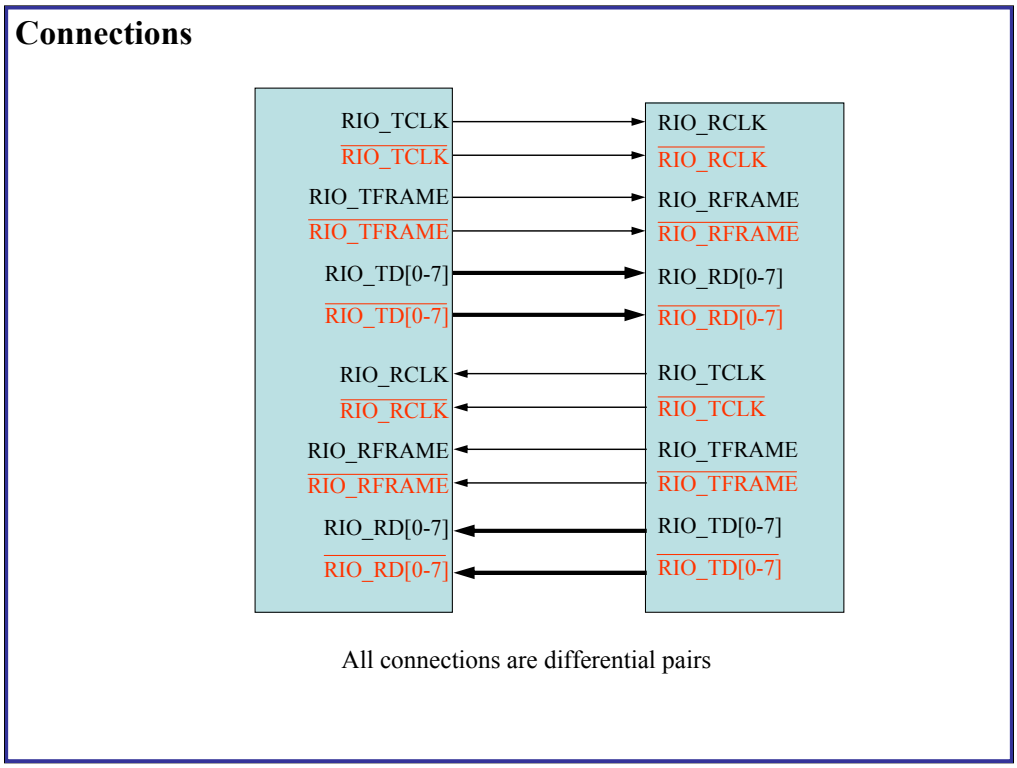


This animation is a simplified view of how data is received in an inbox.

The RapidIO port is eight bits wide.

As data is received it is transferred into a sixty-four bit asynchronous boundary fifo, and when that is full the block is transferred to the protocol layer.

The protocol layer interprets the header information and checks the CRC value, and the data is transferred to the data buffer defined by the receiver.



This diagram indicates the connections required for the RapidIO ports on the PowerQUICC III.

There is one output port and one input port, and all signals are differential pairs, shown here as one black connection and one red, for each signal.

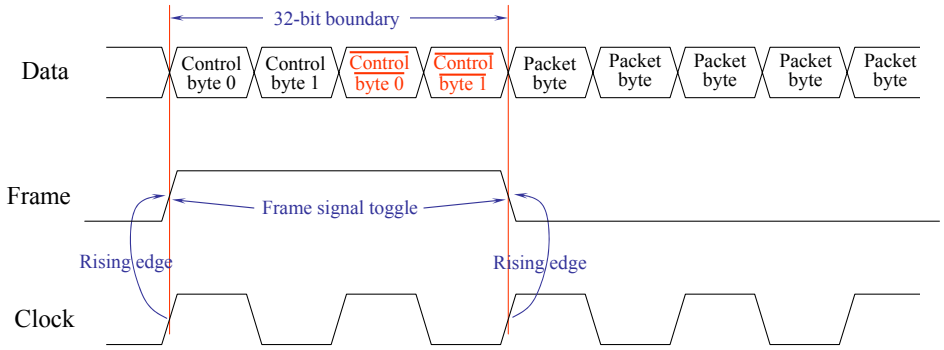
The transmit clock is connected to the receive clock of the other endpoint to maintain data synchronization.

The transmit frame is connected to the receive frame to identify symbol and packet boundaries,

and there are eight data connections.

With both a transmit and receive port this results in a total of forty connections for the RapidIO ports.

Packet start and control symbol delineation



Control framing delineates the start of a packet of the ftype or ttype format.
This only relates to control symbols, not to others such as those containing packet headers and data bytes.
It does toggle for idle control symbols between packets.

This simplified timing diagram indicates the timing relationship of clock, frame and data byte signals at the transmitter. In the receiver the clock is delayed by ninety degrees so that the clock edge aligns to the center of the data.

All data transfers are aligned to thirty-two bit boundaries.

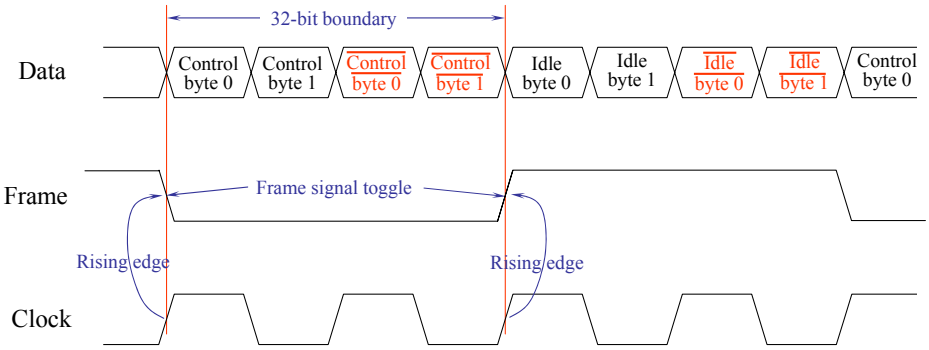
Framing is defined by the frame signal toggling, so every time the frame signal changes state, it indicates a frame boundary. The level of the frame signal holds no specific information, so for this example it's possible the signal could have been inverted and still represent the same information.

This example indicates two bytes of control symbol. Control symbols are transferred with a bit inverted version of the same value to provide hardware error checking. If the inverted version does not provide the correct bit values then the receiver will signal an error.

In this case the frame signal toggles to indicate the start of the control symbol and then toggles again to indicate the start of an ftype or ttype packet.

Packet start and control symbol delineation

Control symbol delineation example

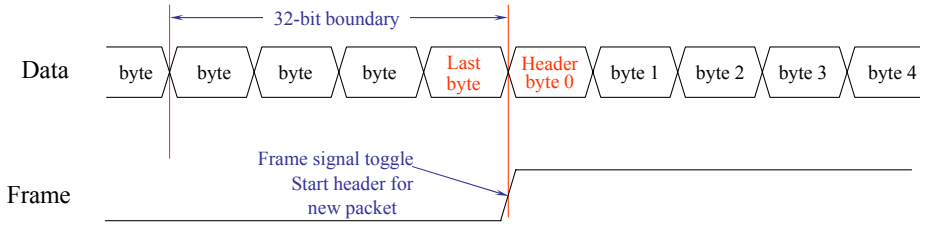


In this example there is a control symbol followed by an idle state, and then another control symbol.

The frame signal toggles for the start of the control symbol, for the start of the idle, and again for the next control symbol. Had there been more idles then the frame would not toggle until the next control symbol or framing event.

Packet termination

Header marked end of packet



Packets may also be terminated with end-of-packet, restart-from-retry, link-request, or stomp control symbol.

A receiver drops a cancelled packet without generating errors and responds with a packet-retry acknowledgement control symbol.

EOP is required if no packet follows (idles only) .

This example indicates a header marked end of packet. In other words the frame signal toggles when a new header is recognized, which is immediately following the end of the previous packet, and so signals to the controller that the previous packet is complete.

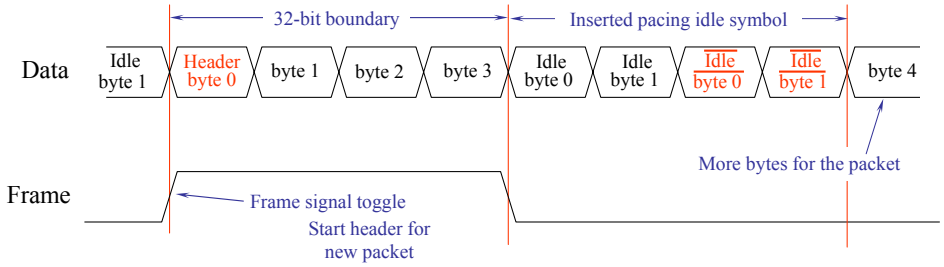
Packets may also be terminated with end-of-packet, restart-from-retry, link-request, or stomp control symbol.

If for any reason the receiver cannot service a packet it simply cancels it and responds with a packet retry acknowledgement. No error is generated. The transmitter is aware that the packet was not processed because of this acknowledgement and resends the packet.

If there are no further packets to transmit at this time then the transmitter transmits and end of packet symbol.

Packet pacing

Pacing idle insertion in packet



The receiver may request the sender to insert pacing idle control symbols by sending a throttle control symbol with a specified number of aligned idle pacing control symbols.

These are not included in the CRC calculation for the packet.

Packet pacing is controlled by inserting idle symbols into the packet stream. If the transmitter receives a throttle control symbol from the receiver, which indicates that the data rate should be slowed, the transmitter inserts the specified number of idle symbols into the packet stream.

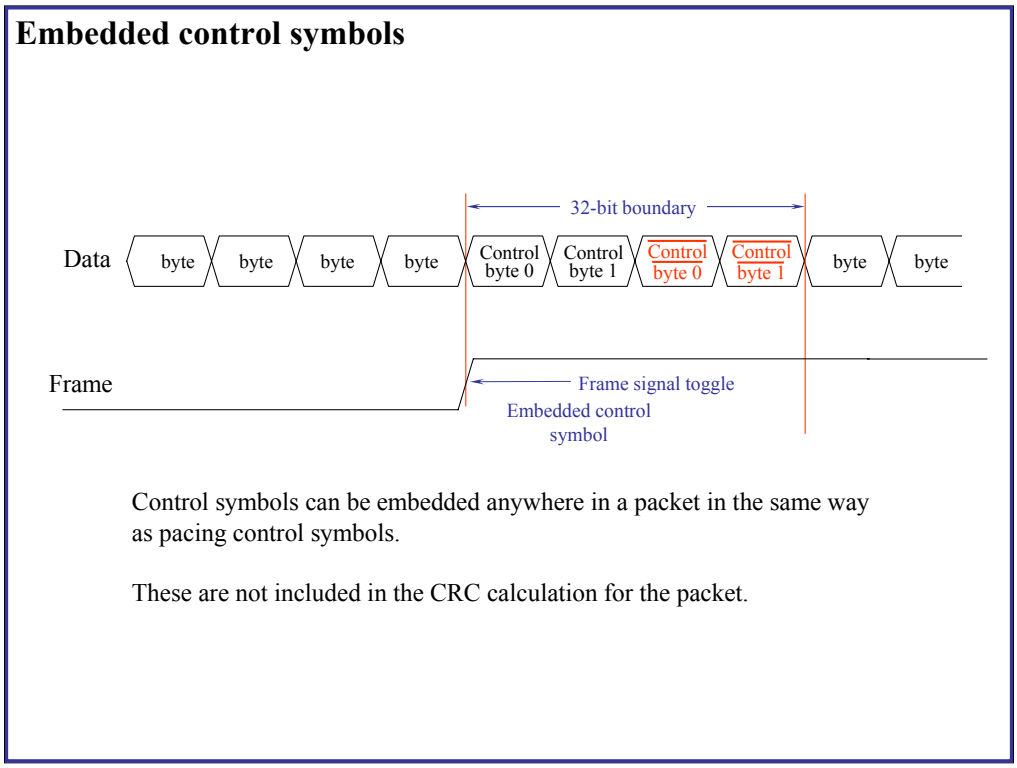
This example shows a single idle symbol inserted after the first four bytes of the packet, and these will always be inserted on a thirty-two bit boundary.

The framing signal toggles for the start of the packet, and also for the start of the idle symbol. Notice that it does not toggle for the next part of the packet. However, if a subsequent idle symbol was added after more data in the packet then the frame signal would again toggle at the start of the next idle symbol.

There could be more than one idle symbol inserted at any point depending on the number requested.

For pacing the idle symbols are not included in the packet CRC calculation.

As in all cases the idle symbol consists of sixteen bits of symbol followed by the inverse bit value.



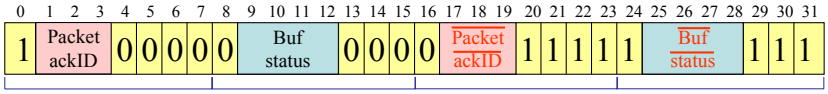
Control symbols can be embedded anywhere in a packet in the same way as pacing control symbols.

These are not included in the CRC calculation for the packet.

The frame signal toggles at the start of the control symbol.

Acknowledgment control symbols - Packet-accepted

Acknowledges that the receiver has accepted the packet indicated and the transmitter is released.
Only sent when a complete, error free packet has been received.
Device cannot issue more than eight unacknowledged packets



Buf status = 1111 Buffer status is not reported

Packet ackID originates from the packet header PMR

Buf status normally returns an indication of how many buffers are available for flow control, enabling transmitter to modify transfer rate if necessary.

Every packet causes an acknowledge control symbol with the corresponding AckID

Devices return acknowledge control symbols in same order as received

As indicated earlier there are control symbols transferred between end points of which there are several different types, and the following slides indicate what they do and their composition.

The operating sequence demonstrated that for every transaction an acknowledgment is returned, and this diagram shows the packet accepted symbol.

The bits shown in yellow are pre-defined, and as you can see the latter half shows a bit inversion of the first half.

Two pieces of information are contained within the symbol, the packet identification number that relates to the packet it relates to, which originates from the received packet header, and buffer status.

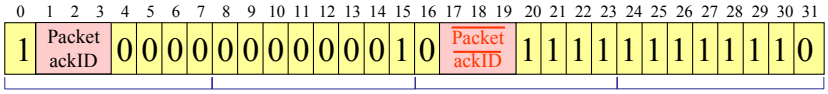
The RapidIO specification for buffer status provides the means of indicating how many buffers are available to enable flow control, but in the PowerQUICC III the value is always set to “1111” since buffer status is not reported.

Although it’s possible for packets to be received out of order due to some error condition, AckID symbols must be returned in order. If they are not this results in an error.

A device cannot issue more than eight unacknowledged packets, and so if this should occur then an error will be generated.

Acknowledgment control symbols - Packet-retry

Signals that the receiver was unable to accepted the packet indicated due to a temporary resource conflict and the transmitter should resend it.
This can be sent any time after the start of a packet, allowing the possibility of reducing bandwidth wastage, by avoiding transmission of rejected packets.



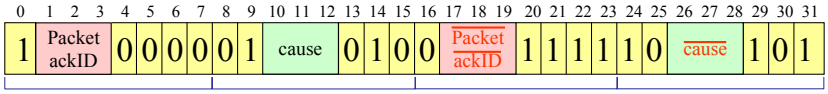
The packet retry symbol signals that the receiver was unable to accept the packet due to a temporary resource conflict and so the transmitter should resend it.

If the conflict should occur soon after the reception of the packet starts this symbol can be sent as soon as resources allow to reduce bandwidth wastage.

Again the yellow bits are predefined, only the packet identification value is returned so that the transmitter is aware of which packet it relates to.

Acknowledgment control symbols - Packet-not-accepted

Signals that the receiver was unable to accepted the packet indicated due to an error and the transmitter should resend it.
This can be sent any time after the start of a packet, allowing the possibility of reducing bandwidth wastage, by avoiding transmission of rejected packets.



Cause	Definition
000	Internal error
001	Received unexpected ackID on packet
010	Received error on control symbol
011	Non-maintenance packet reception is stopped
100	Received bad CRC on packet
101	Received S-bit parity error on packet/control symbol
110	Reserved
111	General error

This indicates that the packet was not accepted due to an error and must be resent.

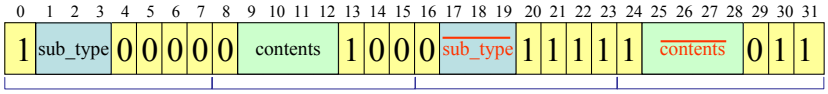
In this case the cause of rejection is indicated, which could be due to some internal error in the receiver, an unexpected identification value, control symbol error, only maintenance packets are being received at present, either a packet CRC error or header S bit parity error was recognized, and general error simply relates to any other error.

In particular this highlights that the receiver tracks packet identification and knows what to expect.

The control symbol error results from the bit inversion not matching the first sixteen bits.

Packet control symbols format

Packet control symbols provide packet delineation, transmission, pacing, and other link interface controls. They include throttle, stomp, restart-from-retry control symbols, idle, end-of-packet, and multi-cast-event control symbols. The content field is dependent on the type of control symbol.



sub_type	code	content
idle	000	Number of max-sized packets that can be received
stomp	001	unused, (0000)
eop	010	Number of max-sized packets that can be received
restart_from_retry	011	unused, (0000)
throttle	100	Number of pacing idles to insert in packet *
Multicast_event	101	Number of max-sized packets that can be received
Reserved	110	
Reserved	111	

* Binary power scale to 1024
1110 = align to drift
1111 = stop idles

Packet control symbols provide packet delineation, transmission, pacing, and other link interface controls. They include throttle, stomp, restart-from-retry control symbols, idle, end-of-packet, and multi-cast-event control symbols.

The content field is dependent on the type of control symbol.

The table indicates the type of control symbol and the content information for each.

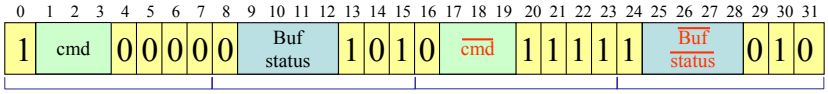
Notice that for the stomp and restart-from-retry the content is not used and is cleared.

In the case of throttle control the content value returns the number of idle symbols to insert as a binary power scale, as well as an indication to align-to-drift to synchronize the clock, and to stop inserting idles.

Link maintenance control symbols - Link-request

To maintain links, link_request control symbols issue commands or request status.
They cannot be embedded in a packet.

Link request training control symbols are followed by at least one idle symbol.



cmd	code	content
Send training	000	Send 256 iterations of training pattern
Reserved	001	
Reserved	010	
Reset	011	Reset receiving device
Input-status	100	Return input port status (restart from error)
Reserved	101	
Reserved	110	
Reserved	111	

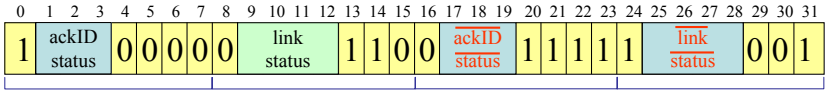
To maintain the connection links the link request issues commands or requests status for the system, offering three possibilities.

To synchronize the link the send training command is used, the receiving device can be reset, or status information for the port can be requested.

These cannot be embedded in a packet, and must be followed by at least one idle symbol.

Link maintenance control symbols - Link-response

Devices responded to a link-request with link_response symbols.
This can be embedded in a packet.



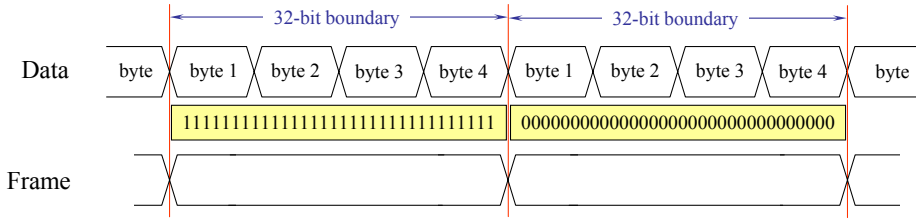
link-status	code	description
Error	0010	Unrecoverable error encountered
retry-stopped	0100	Port has stopped due to retry
error-stopped	0101	Port has stopped due to transmission error
OK. ackID 0 - 7	1000-1111	Device with ID = lsbs working properly
Reserved	all other codes	

ackID status returns the AckID expected

When a link maintenance request is received a link response is returned. This can indicate an unrecoverable error, stopped due to a retry, or a transmission error, or the device is working properly, in which case the device number is provided in the lower significant three bits of the code. For example if this was the response indicating that device three was o.k. then the link status value would be “1011”, where the least significant three bits define the device number.

The acknowledge I.D. status indicates the I.D. of the next expected packet.

Training pattern



A training pattern is needed to set up the input port to sample information correctly by centering clock to data eye.
This shows the training pattern.

All other activity must be stopped before a training pattern is issued, so it cannot be embedded in a packet.

When the send training link request is received a training pattern is transmitted to align the clock in the receiver to the center of the data to ensure accurate sampling.

The pattern is shown here, and 256 iterations are used.

Before this operation can proceed all other activity must be stopped.

Notice that each half of the pattern is defined by the framing signal.

One feature of RapidIO that should be apparent to you by now is that the interface is never static.

If no control symbols or message packets are transferred then idle symbols are.

Packet Formats

Transaction type	Packet type	Definition
Implementation defined	0	Defined by the device implementation
Reserved	1	
ATOMIC	2	Read-modify-write operation on a specific address
NREAD	2	Read specified address
Reserved	3-4	
ATOMIC test-and-swap	5	Not implemented on PQIII
NWRITE	5	Write specified address
NWITE_R	5	Write specified address, notify source of completion
SWRITE	6	Write specified address (DW only - simpler header)
Reserved	7	
MAINTENANCE	8	Read or write device configuration registers and perform other system maintenance tasks
Reserved	9	
DOORBELL	10	Send a short message (interrupt)
MESSAGE	11	Send a message
RESPONSE	13	

Now that the control symbols have been identified the packet formats can be discussed.

This table shows all the packet types defined by the RapidIO specification, and those supported by the PowerQUICC III.

Type two packets are the ATOMIC, which is a read modify write operation on a specific address, and NREAD transaction which is a read of a specific address.

For example if the processor, or a controller within the PowerQUICC III, requests a read of data from an address that happens to align to the RapidIO space the controller will provide an NREAD transaction, requesting data from an external device that it is connected to.

Type five packets relate to either an NWRITE transaction, which writes data to a specific address, or an NWRITE with response, which does the same thing but also returns a notification that the write has completed.

Type six packets are similar, but simpler than type five in that they perform a streaming write to a specified address, which always transfers double words of thirty-two bits. Because they are always aligned less information is required in the header to define the data size and location.

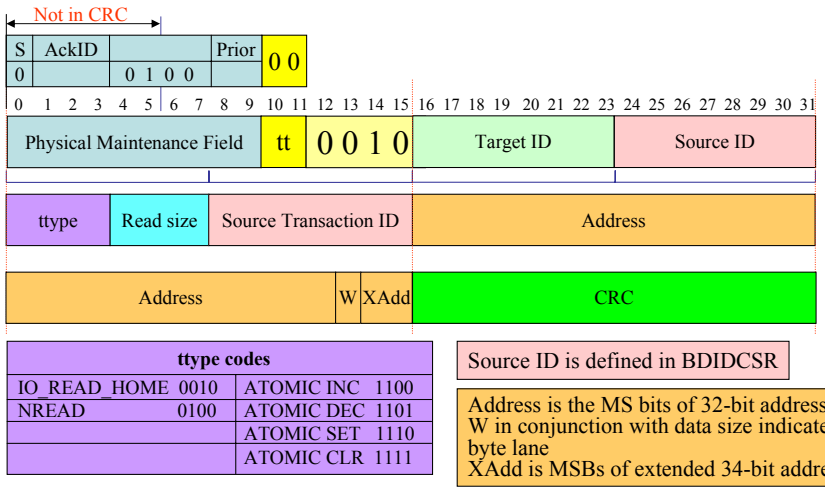
Type eight packets are for maintenance, either reading or writing configuration registers or system maintenance tasks.

Type ten packets are doorbells. These offer a similar function to their namesake, effectively a request for attention. They contain sixteen bits of information which can be useful to an interrupt service routine.

Type eleven packets are messages, and type thirteen are the response to a message.

Type 2 packet format

Requests a read-modify-write operation or a normal read operation



This diagram shows the format for a type two packet, which is indicated in bits twelve through fifteen.

The first ten bits provide the physical maintenance field, shown in more detail above. Some bits are pre-defined but three bits provide the acknowledge I.D. for tracking the transaction acknowledgement. These are a simple number which is automatically incremented for each packet transmitted.

The PMF also provided the packet priority .

The first six bits of the PMF are not used in the CRC calculation for the packet, having their own parity check.

tt is the transport field which is always zero for the PowerQUICC III which uses the small size transport size, providing device I.D. numbers up to 255.

Target I.D. indicates the destination device, and the Source I.D. identifies which end point originated the transaction. The source I.D. is provided by the architectural **B**ase **D**evice **I**.**D**. **C**ommand and **S**tatus **R**egister, and originates from hardware configuration.

ttype provides the actual read type code used by the receiver controller to provide the correct operation.

Read size defines the data size of the operation.

Source transaction I.D. provides identification of this transaction to ensure that the response is correctly associated to the request.

The address field provides the most significant twenty-nine bits of the thirty-two bit address of the data requested.

W is a value which is used by the controller in conjunction with the data size to define the most significant byte lane of the data access.

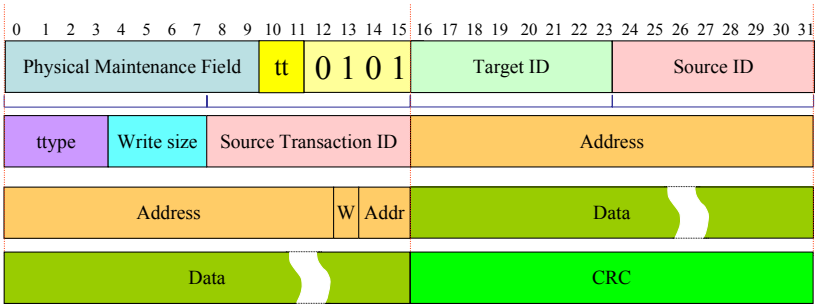
Xadd provides the two most significant address bits of the thirty-four bit RapidIO address.

The last sixteen bits provide the CRC value for the packet.

This packet always consists of twelve bytes.

Type 5 packet format

This is a write packet that always contains a data payload, which could be multiple double-word values, or a single double-word or less. The write size contains a code defining the payload size.



ttype codes	
FLUSH	0001
NWRITE	0100
NWRITE_R	0101

Packets not aligned to 32-bit boundary are padded with 16 bits of '0'.

This is a write packet and so always contains a data payload. The payload consists of at least one double word value and could be multiples.

In all headers bits twelve through fifteen indicate the packet type, in this case five.

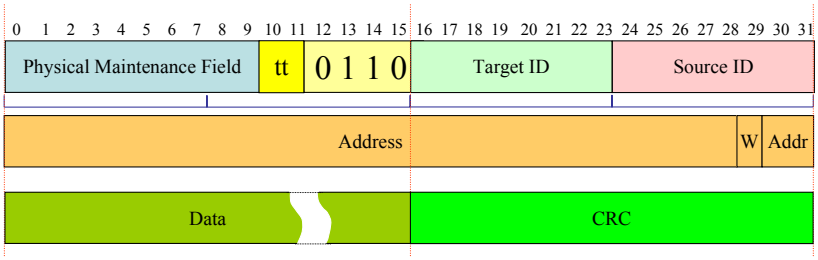
The majority of the fields contain very similar information to the type two packet.

ttype defines the type of write, and the write size defines the data size to write.

The minimum size of data in the packet is thirty-two bits, even if only a single byte is to be written, and padding will be applied as necessary, and the data could be up to sixteen bytes in double word increments. For data not aligned to the thirty-two bit boundary padding is applied to meet the alignment rules.

Type 6 packet format

This is a special purpose packet called a streaming write, and is only used for the SWRITE transaction. It always contains a data payload, with a minimum of one double-word and up to 256 bytes.
Since it only relates to one transaction type the ttype size and transaction ID are not required.


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Physical Maintenance Field
tt
0 1 1 0
Target ID
Source ID
Address
W
Addr
Data
CRC

The type six packet is simply a variation of the type five packet, providing a simpler function for what is called a streaming write transaction.

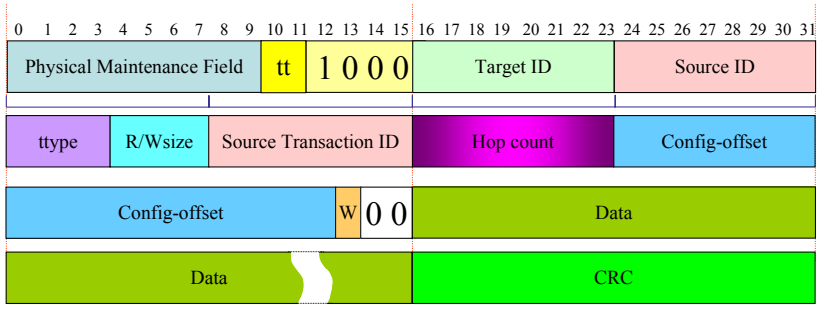
SWRITE is a double word only transaction, and so requires less information for data handling.

Neither NWRITE nor SWRITE transactions receive a response , so there is no notification to the sender that the data has been handled by the receiver, although the acknowledgement from the next port does indicate that the data has been successfully transferred to the fabric. Had the transaction not been successfully transferred on then the acknowledgment process should ensure a retry.

NWRITE_R however does provide a response.

Type 8 packet format - maintenance request

Maintenance packets used for accessing CARs, CSRs, and data structures, and serves for both request and response.
Only contains data payloads for both read and write accesses, and registers are always accessed in 4-byte or 8-byte multiples up to a maximum of 64-bytes, while data transfers are 8-byte multiples.



ttype codes	
READ	0000
WRITE	0001
PORT WRITE	0100

Hop count = distance into network to access
Config_offset = double_word offset into register space

Maintenance packets are used for accessing architectural registers and data structures, and serves for both request and response.

They only contains data payloads for both read and write accesses, and registers are always accessed in 4-byte or 8-byte multiples up to a maximum of 64-bytes, while data transfers are 8-byte multiples.

These transactions can be for the switch fabric, which have the same memory map structure as the end points. Since there is no mechanism for directly addressing a switch the location for this transaction is defined by the number of connection hops required to reach the defined switch, and configuration offset indicates the offset into the memory map to the destination.

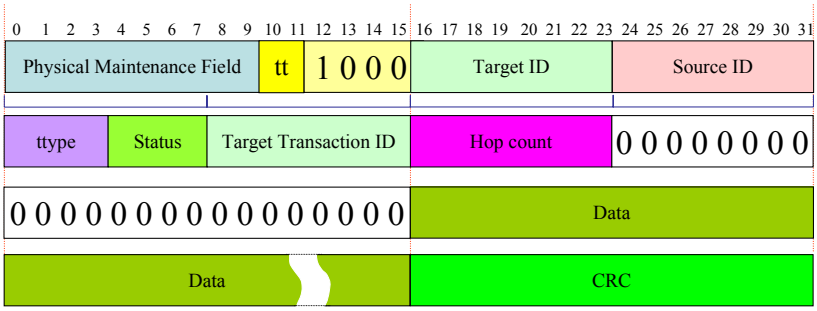
For an example of this click on the hop count field.

The type of transaction requested is defined by ttype defining either a read, write, or port write transaction. The read and write transactions can be specified as four byte, eight byte double words, or multiple double word quantities up to sixty-four bytes. Both of these transactions generate the appropriate maintenance response.

The port write does not guarantee delivery and has no associated response, and can be used for error indicators or status information for switches.

Type 8 packet format - maintenance response

Maintenance packets used for accessing CARs, CSRs, and data structures, and serves for both request and response.
Only contains data payloads for both read and write accesses, and registers are always accessed in 4-byte or 8-byte multiples up to a maximum of 64-bytes, while data transfers are 8-byte multiples.


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Physical Maintenance Field
tt
1 0 0 0
Target ID
Source ID
ttype
Status
Target Transaction ID
Hop count
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Data
Data
CRC

ttype codes	
READ resp	0010
WRITE resp	0011

STATUS codes	
DONE	0000
ERROR	0111

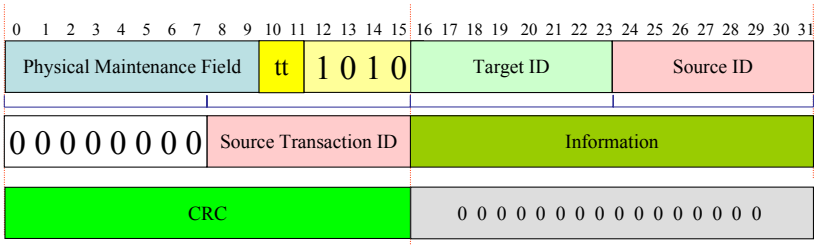
The response packet indicates the type of transaction being responded to in the ttype field, and the status of the transaction.

If this is the response to a read then the data for the read is in the packet.

For a write there is no data in the response packet.

Type 10 packet format - doorbell

Doorbell packets are used to generate interrupts if enabled.
They never contains data payloads, and the information field is software defined.

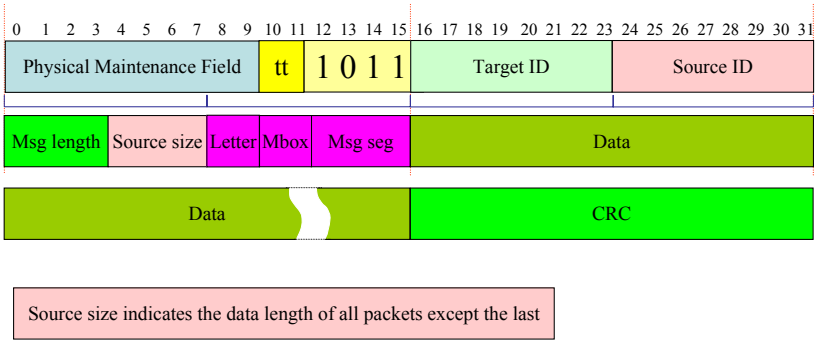


Doorbell packets are used to generate interrupts if enabled.

They never contain data payloads, and the information field can be utilized by software to provide information for the interrupt service routine.

Type 11 packet format - message

Message packets always contain double word data payloads.
Any data not specified as multiples of 8-bytes must be handled in software.
Letter, mailbox, and message segment uniquely identifies the message.



Source size indicates the data length of all packets except the last

Letter, Mbox, Msg seg could be used to define one of 256 single segment mailboxes

Message packets always contain double word data payloads, and any data not specified as multiples of eight bytes must be handled in software, since all data must be a double word multiples.

Each packet can transfer up to 256 bytes of data for a message segment, and a message can consist of up to sixteen segments.

The message length defines the total number of segments of the message to be sent.

The source size defines the length of data within the packet and relates to all segments of the message. Using this value, knowing the length of the data in all packets except the last, which will probably be less, the receiver can calculate the location of the data for each packet regardless of the order received.

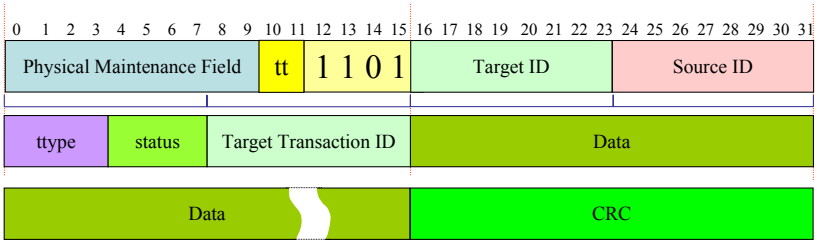
Letter, mailbox, and message segment uniquely identifies the message. The message segment is the count of each segment sent out which enables the receiver to determine which segments have been received, how many more are expected, if any have been lost or have been received out of order, and combined with the source size can calculate the address of the destination memory.

Letter and mailbox are from by the RapidIO specification, but actually have little meaning with respect to the PowerQUICC, since only one mailbox and one letter are used. There is no way of distinguishing these even though they can be specified.

When dealing with the PowerQUICC III be aware of the restrictions regarding mailbox and letter assignments, but since there is only one queue there is no way of utilizing the designations provided.

Type 13 packet format - response

Response packets are always returned when a request has been completed. They may or may not include data payload.
Errors never have a data payload.
These are used for all requests except maintenance and response-less writes.


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Physical Maintenance Field
tt
1 1 0 1
Target ID
Source ID
ttype
status
Target Transaction ID
Data
Data
CRC

ttype codes	
REQUEST RESP	0000 (NWRITE)
MESSAGE RESP	0001
REQUESTRESP+DATA	1000 (NREAD)

Target Tran ID = letter(2), mbox(2), mseg(4)

STATUS codes	
DONE	0000
DATA ONLY	0001
RETRY	0011
DONE-INTERVENTION	0101
ERROR	0111
USER	1100-1111

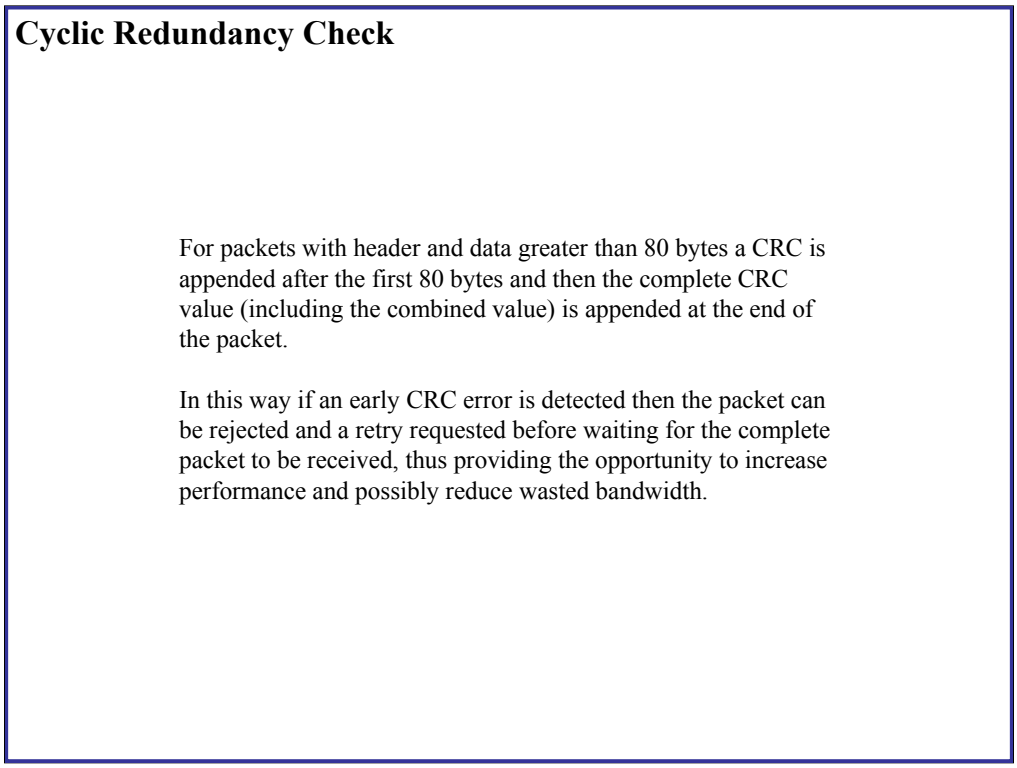
Response packets are always returned when a request has been completed. They may or may not include data payload.

Errors and requests not expecting it never have a data payload.

This type thirteen response applies for all requests except maintenance and response-less writes. The ttype field indicates which type of transaction the response relates to and the target I.D. and transaction I.D. specifies which transaction. The transaction I.D. returns the letter, mailbox, and message segment of the requested message the response relates to.

The status field provides the result of the transaction

For further information on the packet formats look in the RapidIO Interconnect specification Part one : Input Output logical specification, section three.


Cyclic Redundancy Check
For packets with header and data greater than 80 bytes a CRC is appended after the first 80 bytes and then the complete CRC value (including the combined value) is appended at the end of the packet.
In this way if an early CRC error is detected then the packet can be rejected and a retry requested before waiting for the complete packet to be received, thus providing the opportunity to increase performance and possibly reduce wasted bandwidth.

For simplicity earlier diagrams only indicated a single CRC value at the end of the packet.

However, for packets with header and data greater than 80 bytes a CRC is appended after the first 80 bytes and then the complete CRC value (including the combined value) is appended at the end of the packet.

In this way if an early CRC error is detected then the packet can be rejected and a retry requested before waiting for the complete packet to be received. This is most useful for switches in “cut through mode” providing the ability of an early check that the packet is heading in the correct direction.

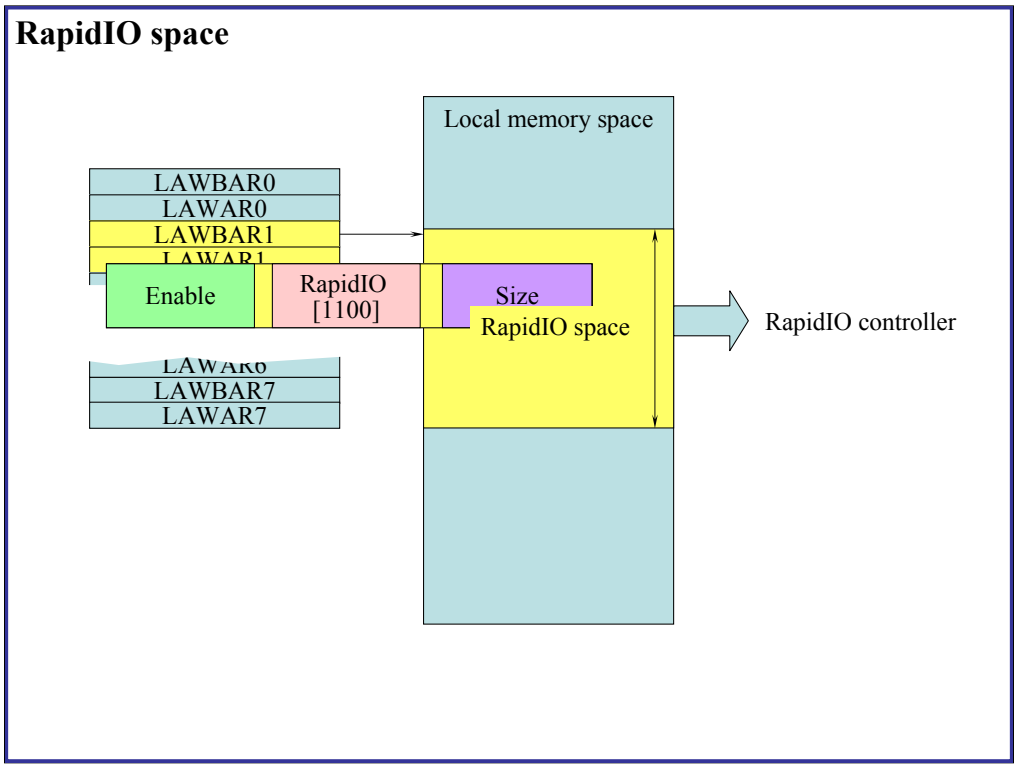
Reserved field behavior for PQIII

Reserved packet fields are assigned ‘0’
Reserved packet formats are not used
Reserved packet fields are ignored
Reserved encodings are never assigned to packet fields
Implementation-defined packet fields are ignored unless understood by by receiver
Received reserved packet field encodings are ignored if it is not necessary for the field to be defined for the requested transaction
Reads of reserved register* fields return ‘0’ and do not cause an error
Writes to reserved register* fields are ignored and do not cause an error
Reads of implementation-defined registers* and extended features return defined values
Writes to implementation-defined registers* and extended features are determined by the implementation

* Registers refer to the CARs and CSRs

Reserved fields are dealt with in an orderly and un-intrusive manner in the PowerQUICC III.

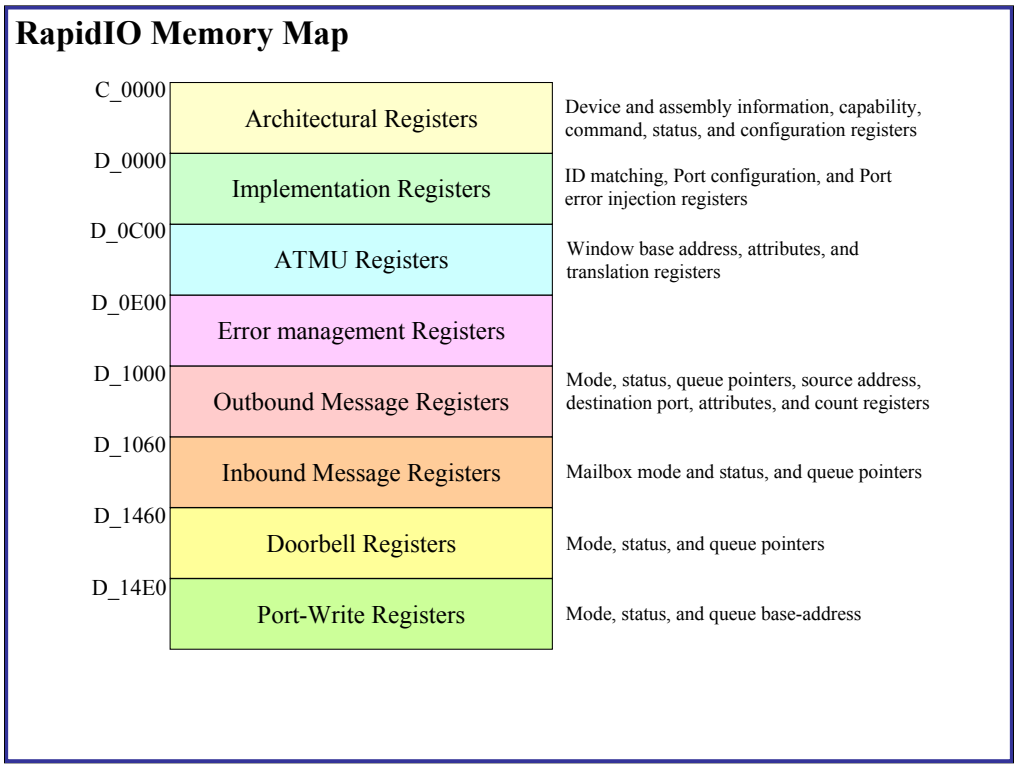
This is a simplified chart of how they are handled.



An important consideration is how does information related to the RapidIO ports relate to the rest of the PowerQUICC III ? Lets consider where the core requests some data. How does the system know if that data is out on the RapidIO port or elsewhere ? Within the PowerQUICC III is a bank of registers known as the Local Access Window registers, which are arranged in pairs.

Each of these pairs provide the definition of what areas of local memory space are allocated to which controller. These are the local access windows.

The Local access window base address window provides the address of the start of the address space, and the attributes register indicates if this window is enabled, which controller uses this space and how large is the memory in this window. Any internal transactions that take place within the PowerQUICC III are compared to the law registers and if there is a match then the transaction is applied the defined controller. In the example suggested here, when the core requests data from a given address, if that address lies within the RapidIO window then it will be converted to a RapidIO read transaction, and the response will be transferred back to the core.



The RapidIO controller has it's own memory map shown here. These locations are at predefined offsets within the RapidIO space.

This diagram is not drawn to scale and there are varying numbers of registers within the different blocks.

The architectural registers provide device and assembly information, capability, command, status, and configuration definition.

Implementation registers provide I.D. matching, Port configuration, and Port error injection information.

ATMU registers provide window base address, attributes, and translation registers for access into the RapidIO specific space.

Error management registers provide information on errors and control functions for error handling.

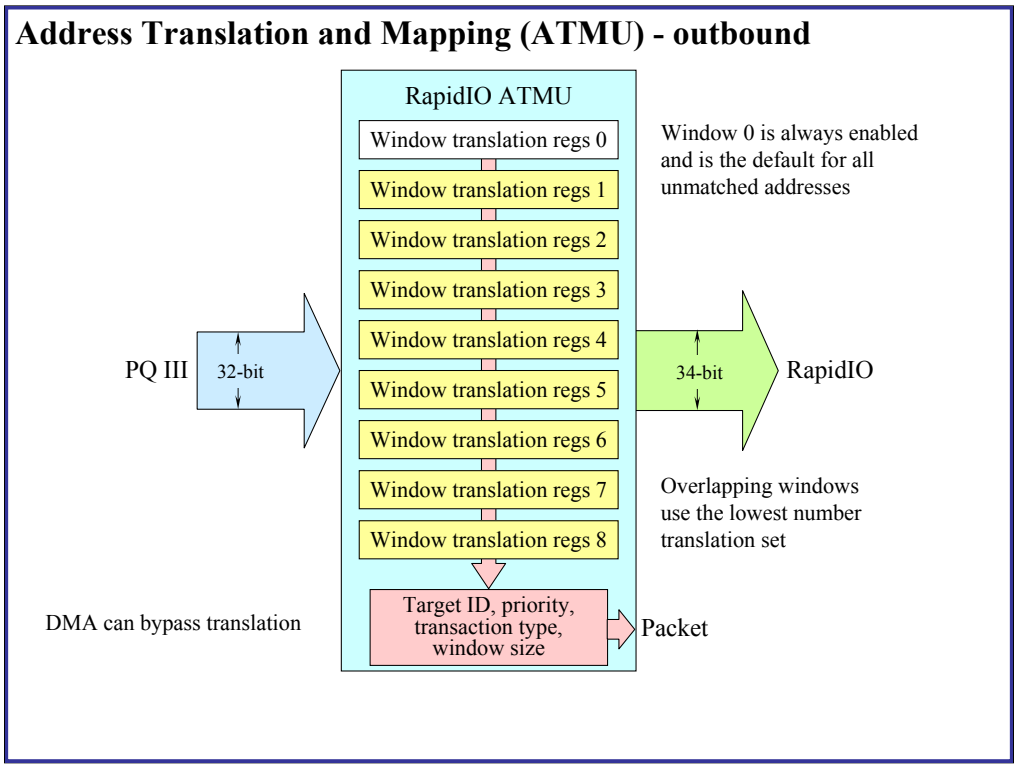
Outbound message registers provide mode, status, queue pointers, source address, destination port, attributes, and counters for outbound messages.

Inbound message registers provide mailbox mode and status, and queue pointers for inbound messages.

Doorbell registers provide mode, status, and queue pointers for Doorbell packets.

Port write registers provide mode, status, and the queue base-address for Port write packets.

All of this map must be initialized as required for the RapidIO controller to operate correctly.


Address Translation and Mapping (ATMU) - outbound
RapidIO ATMU
Window translation regs 0
Window translation regs 1
Window translation regs 2
Window translation regs 3
Window translation regs 4
Window translation regs 5
Window translation regs 6
Window translation regs 7
Window translation regs 8
Window 0 is always enabled and is the default for all unmatched addresses
PQ III
32-bit
34-bit
RapidIO
Overlapping windows use the lowest number translation set
DMA can bypass translation
Target ID, priority, transaction type, window size
Packet

Address translation and mapping provides address translation from the thirty-two bit internal address into the RapidIO thirty-four bit addressing and translates the requirements of the transaction to RapidIO functions.

For the outbound transactions there are nine possible windows available to the PowerQUICC III RapidIO interface.

Window zero is the default, and any address arriving at the controller that does not match any other window is routed to this window.

Any outbound address appearing from the ocean switch fabric and matching the RapidIO space is compared with the window translation registers, and if there is a match then the translated address is passed on to the RapidIO controller as a thirty-four bit address and supplies target identity, priority, transaction type, and window size.

Most transactions use this mechanism, but the DMA controller can bypass translation.

Special considerations for outbound transactions

Outbound IO_READ transactions cannot cross a 32-byte cache-line boundary, and are set to 32-byte aligned in order to fetch an entire cache-line.

Transactions originated by PCI must map to a window with flow level zero and the PCI bit set.

Maintenance transactions may not exceed 64-bytes.

Doorbells have 2, 4, or 8 byte transactions aligned to double word boundaries.

SWRITE must be a minimum of 8-bytes.

ATOMIC transactions are limited to 1, 2, or 4 bytes and must meet the RapidIO specification

Outbound I/O read transactions cannot cross a thirty-two byte cache line boundary, and are set to thirty-two byte aligned in order to fetch an entire cache-line.

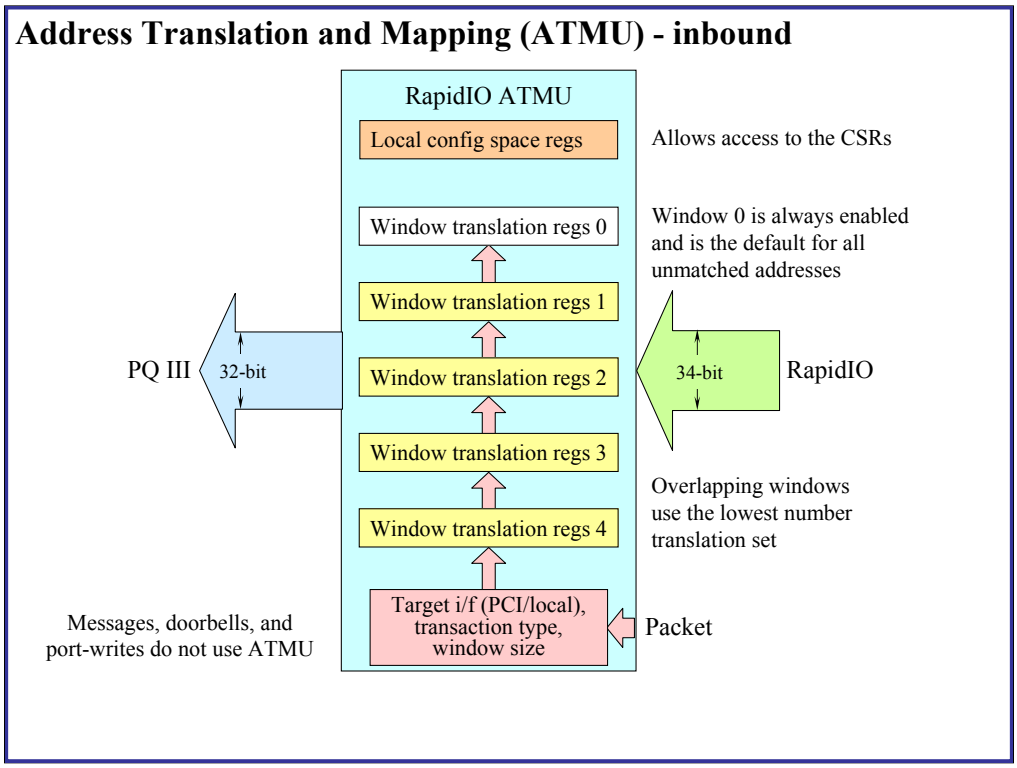
Transactions originated by PCI must map to a window with flow level zero and the PCI bit set.

Maintenance transactions may not exceed sixty-four bytes.

Doorbells have two, four, or eight byte transactions aligned to double word boundaries.

SWRITE must be a minimum of eight bytes.

ATOMIC transactions are limited to one, two, or four bytes and must meet the RapidIO specification


Address Translation and Mapping (ATMU) - inbound
RapidIO ATMU
Local config space regs
Allows access to the CSRs
Window translation regs 0
Window 0 is always enabled and is the default for all unmatched addresses
Window translation regs 1
Window translation regs 2
Window translation regs 3
Window translation regs 4
Overlapping windows use the lowest number translation set
PQ III
32-bit
34-bit
RapidIO
Target i/f (PCI/local), transaction type, window size
Packet
Messages, doorbells, and port-writes do not use ATMU

Inbound address translation and mapping provides the reverse process of the outbound ATMU .

In this case inbound messages have their thirty-four bit address translated to the local thirty-two bit address, and transfer the transaction to either the PCI controller or the local bus. Remember that in this case local bus means everthing that is connected to the PowerQUICC III main bus. It does not have the same connotation as for previous members of the family.

For inbound transactions there are five translation windows, and window zero is again the default, where any unmatched address is handled.

There are also configuration space registers to allow access to the control, command, and status registers of the RapidIO controller.

Messages, doorbells, and port-writes do not use the ATMU since they have their own mechanism for dealing with transferring data.

Special considerations for inbound transactions

IO_READ_HOME and FLUSH_WITH_DATA transactions always enable snooping and must target local memory.

Maximum data for an IO_READ_HOME is 32 bytes.

PCI reads must have a priority of zero and writes a priority of one.

NWRITE_R transactions must not target PCI.

ATOMIC transactions are limited to 1, 2, or 4 bytes and must meet the RapidIO alignment specification, and target DDR memory.

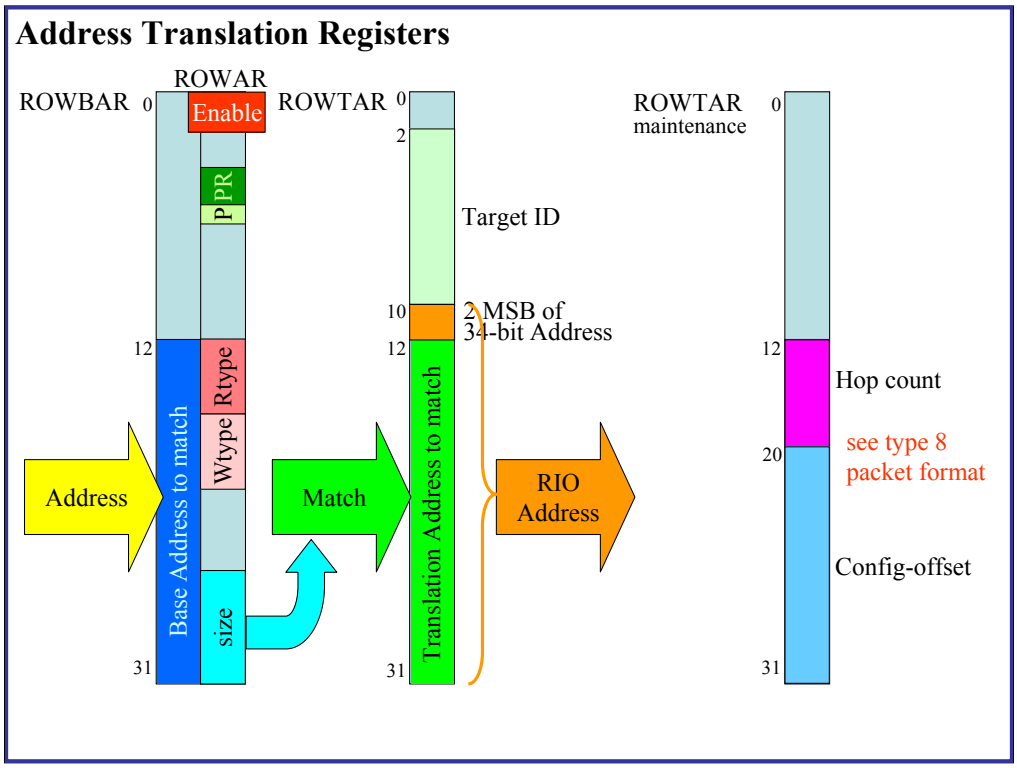
IO_READ_HOME, and FLUSH_WITH_DATA transactions always enable snooping and must target local memory.

Maximum data for an IO_READ_HOME is thirty-two bytes.

PCI reads must have a priority of zero and writes a priority of one.

NWRITE_R, transactions must not target PCI.

ATOMIC transactions are limited to one, two, or four bytes and must meet the RapidIO alignment specification, and target DDR memory.



Outbound address translation is provided with the use of three registers, the **R**apidIO **O**utbound **W**indow **B**ase **A**ddress **R**egister, the **R**apidIO **O**utbound **W**indow **A**ttributes **R**egister, and the **R**apidIO **O**utbound **W**indow **T**ranslation **A**ddress **R**egister.

When an address has matched the local access window that indicates that this transaction is directed to the RapidIO port then that address is then compared to a combination of the base address and attributes registers.

If the address lies within the boundary of the base address and size of one of the pairs for window one through eight, and that window is enabled, then the twenty two most significant bits of the thirty-four bit translation address are provided by the translation address register.

The lower twelve bits are passed directly and so the minimum window size must be four kilobytes.

The translation address register also provides the target I.D. that will be passed to the header.

The attributes address register also defines the priority of the transaction, if it should follow PCI ordering rules, and the type of read or write including maintenance transaction.

If this is a maintenance transaction then the translation register will have a different format, in that case provides the hop count and configuration offset to the header. If the initial address does not match windows one through eight then it will automatically be handled by window zero. Because of that window zero does not have a base register, and all unmatched addresses are accepted.

The result in these circumstances will be whatever the user defines, and so this must be configured accordingly.

Message unit Features

- One inbound data message structure (inbox)
- One outbound data message structure (outbox)
- Support for chaining and direct modes for the outbox
- Maximum 4 Kbytes messages
- Support for up to 16 segments per message
- Support for up to 256 bytes per packet (segment)
- Support for one inbound doorbell message structure
- Support for transmitting and receiving messages to any mailbox
- Device port is only enabled for transmit/receive after both sending and receiving idle control symbols.

The message unit features include the following :-

One inbound data message structure known as an inbox

One outbound data message structure known as an outbox

Support for chaining and direct modes for controlling the outbox

A maximum of four kilobytes per message

Support for up to sixteen segments per message

Support for up to 256 bytes per packet or segment

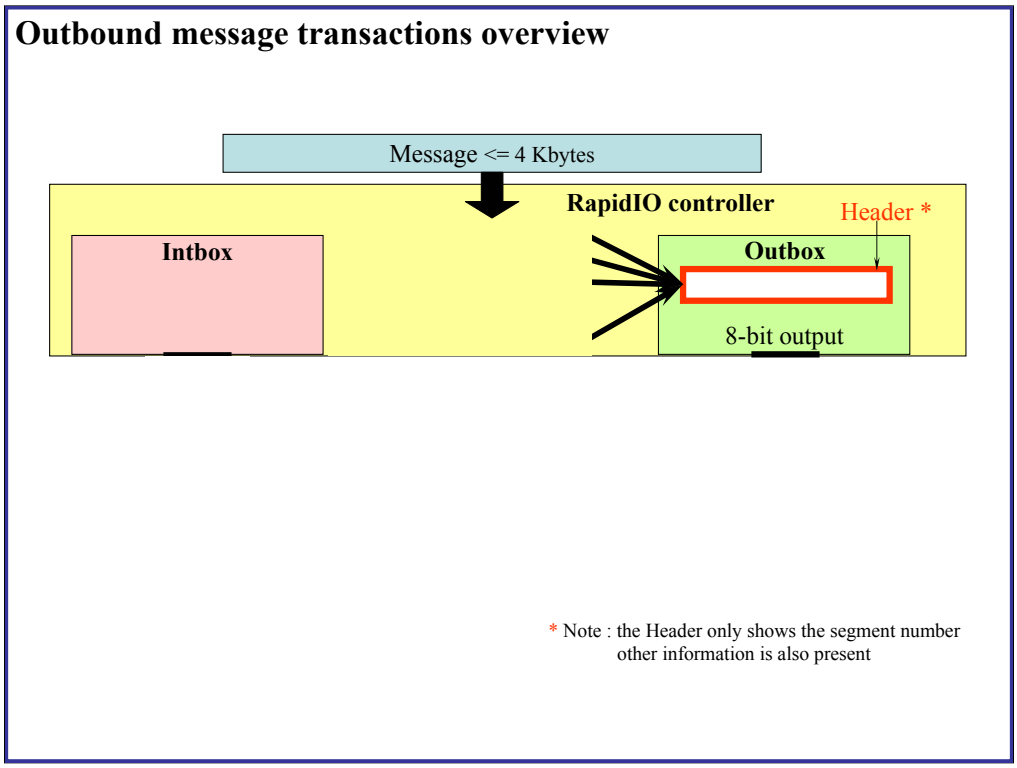
Support for one inbound doorbell message structure

Support for transmitting and receiving messages to any mailbox.

> Be aware that although any allowed mailbox number can be defined, only one mailbox is supported and there is no mechanism to support multiple mailboxes.

Letters are supported, and any letter number can be received, but in the PowerQUICC III DMA is always assigned letter three, and the message unit assigns letter zero. Letters one and two are not available.

The device port is only enabled for transmit/receive after both sending and receiving idle control symbols.



This animation provides a simple overview of the message operation.

The example shows a message that requires transmitting that could be up to four kilobytes in length. The message is handled by the RapidIO controller, which will transfer it through the outbox onto the eight bit output port.

Initially the message is divided into portions of up to 256 bytes of which there can be a maximum of sixteen. All segments are of equal size except the last which could be equal or smaller than the rest.

For efficiency it makes sense to have each segment at the maximum size, but if other requirements dictate then they can be smaller if the message is smaller.

The first segment is transferred to the outbox where the header is attached. Within the header is the segment number which is shown here.

For simplicity other header details are not shown.

The packet is then transmitted out onto the port as a multiple byte stream, and an acknowledgement is received from the other end to indicate that the packet was received at the next stage, which could be either an end point or a switch.

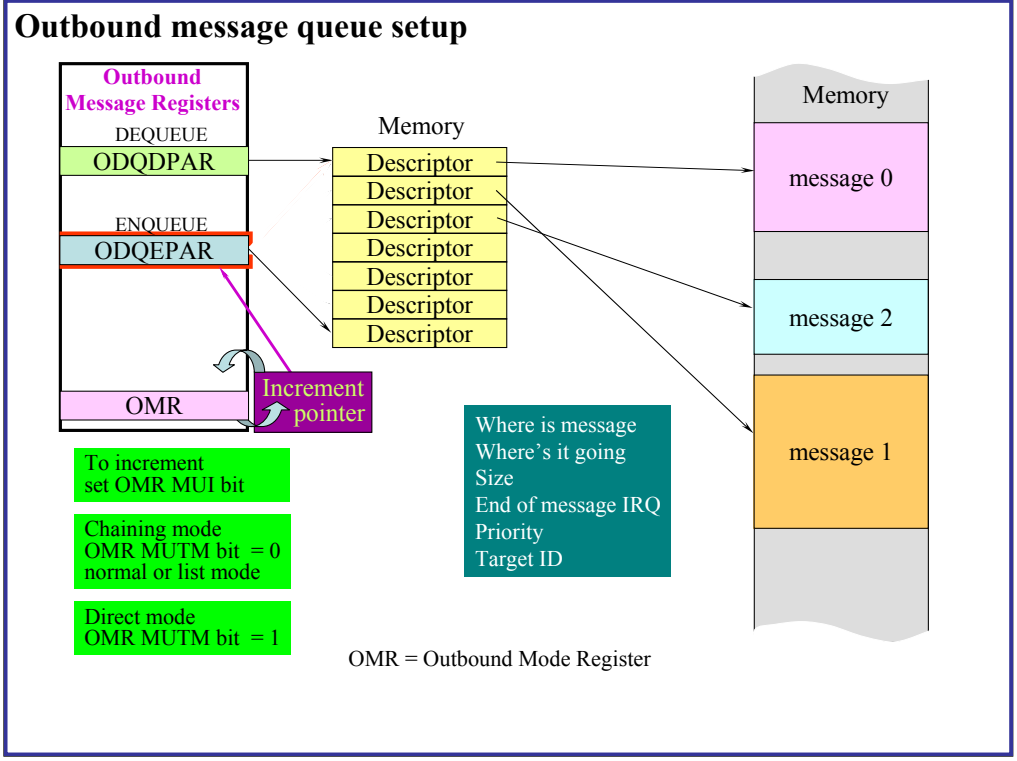
Eventually a response to the packet will be received to indicate that the message was received at the destination.

This process will repeat with the segment number incremented for each packet until the complete message has been transmitted.

It’s important to be aware that the response may not be received before subsequent packets have been transmitted, and it’s even possible that they may be out of order due to retry or error conditions.

However, the acknowledge symbols must be received in order .

Timers are available to check for reception of acknowledgements and responses.



Messages can be queued in memory ready to send. There is a mechanism available to enable this function consisting of some registers and user specified descriptors in memory.

Consider what the controller needs to know to be able to handle the messages :-

Where is the message ? Where is it to be sent ? How big is it ? Do you need to know when it has been sent ? What is it's priority with respect to other messages ? What is the target identity ?

All of this detail is provided by the Queue descriptor, which must be initialized before the message can be sent. The descriptors must be setup as a contiguous chain in memory for queuing multiple messages.

So where is the descriptor, and how does the user set it up ?

There are outbound message registers in the controller, two of which are the DEQUEUE pointer and

ENQUEUE pointer. These are used to define the start of the queue of descriptors and the current or next descriptor to access. They must be initialized to point to the same location in memory where the queue descriptors are to be located.

To initialize the descriptor the ENQUEUE pointer is used to define the location of the descriptor, and the descriptor is loaded with the above mentioned data for the first message.

When the initialization is complete the user must poke a bit in the **O**utbound **M**ode **R**egister which causes the controller to increment the ENQUEUE pointer to the next descriptor location, and this also allows the controller to be aware that the first message is ready to send. That process is described later.

Now the next descriptor can be initialized as before, for the next message, and then the OMR poked to increment the ENQUEUE pointer to the next descriptor.

This process is repeated for as many messages are to be in the queue.

During this process the increment of the ENQUEUE pointer is controlled by setting the MUI bit in the OMR. Once the hardware has incremented the pointer this bit is automatically cleared.

This is the normal mode of operation called chaining mode which is defined by a bit in the OMR designated MUTM when clear.

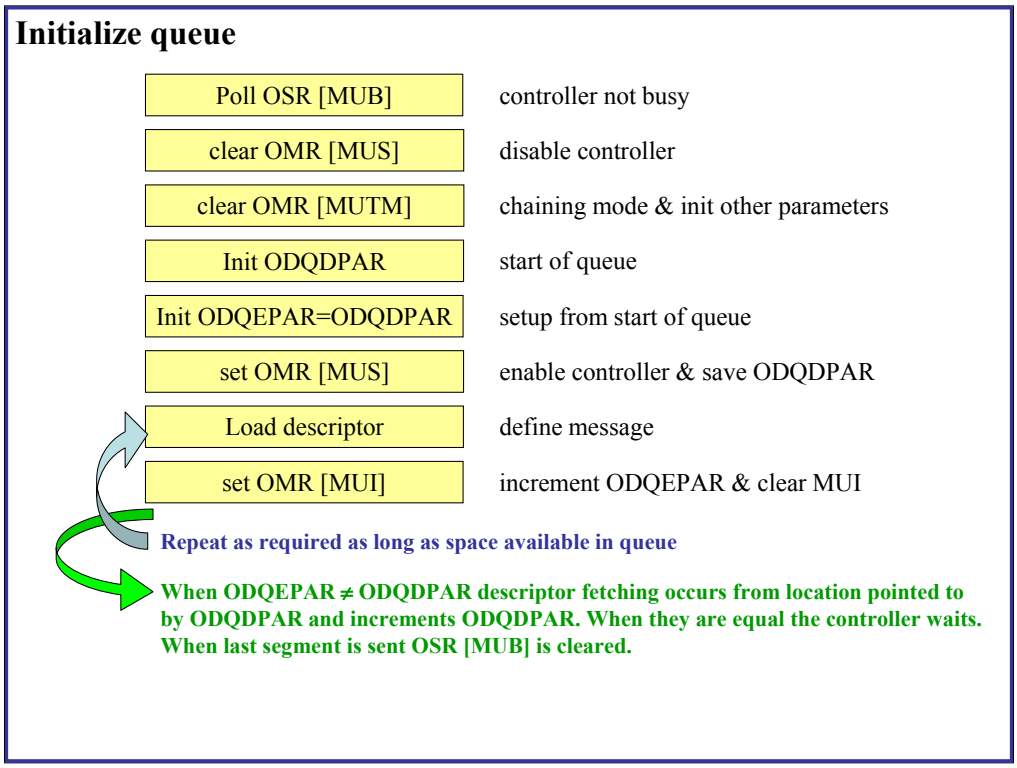
This operation allows automated transfers of messages as the queue is being initialized.

There is another option called list mode, in which the queue is initialized completely before enabling the controller. In that case the process is simpler in that the descriptors are all loaded into memory directly by software and when ready the transfer enabled. It is then the software's responsibility to ensure that everything is set up correctly.

The alternative to chaining is direct mode, defined by setting the MUTM bit.

In this mode descriptors are not used, and all information is provided by the current parameters which are programmed directly into the control registers.

This will be more readily understood when the message processing has been described.


Initialize queue
Poll OSR [MUB]
controller not busy
clear OMR [MUS]
disable controller
clear OMR [MUTM]
chaining mode & init other parameters
Init ODQDPAR
start of queue
Init ODQEPAR=ODQDPAR
setup from start of queue
set OMR [MUS]
enable controller & save ODQDPAR
Load descriptor
define message
set OMR [MUI]
increment ODQEPAR & clear MUI
Repeat as required as long as space available in queue
When ODQEPAR ≠ ODQDPAR descriptor fetching occurs from location pointed to by ODQDPAR and increments ODQDPAR. When they are equal the controller waits. When last segment is sent OSR [MUB] is cleared.

This shows the procedure for initializing the outbound message queue.

First the MUB bit must be polled in the **O**utbound **S**tatus **R**egister to ensure that the controller is not busy.

When not busy the controller must be disabled using the MUS bit in the **O**utbound **M**ode **R**egister.

Now the procedure can be started, and chaining mode is enabled by clearing the MUTM bit in the OMR.

Initialize the DEQUEUE pointer by loading ODQDPAR with the address of the first location of the first descriptor.

Initialize the ENQUEUE pointer ODQEPAR with the same value as the DEQUEUE pointer.

After this set the MUS bit in OMR which causes the controller to save the DEQUEUE pointer internally.

Now the descriptor can be initialized describing the message which is either already in memory or will be when transmission is to begin.

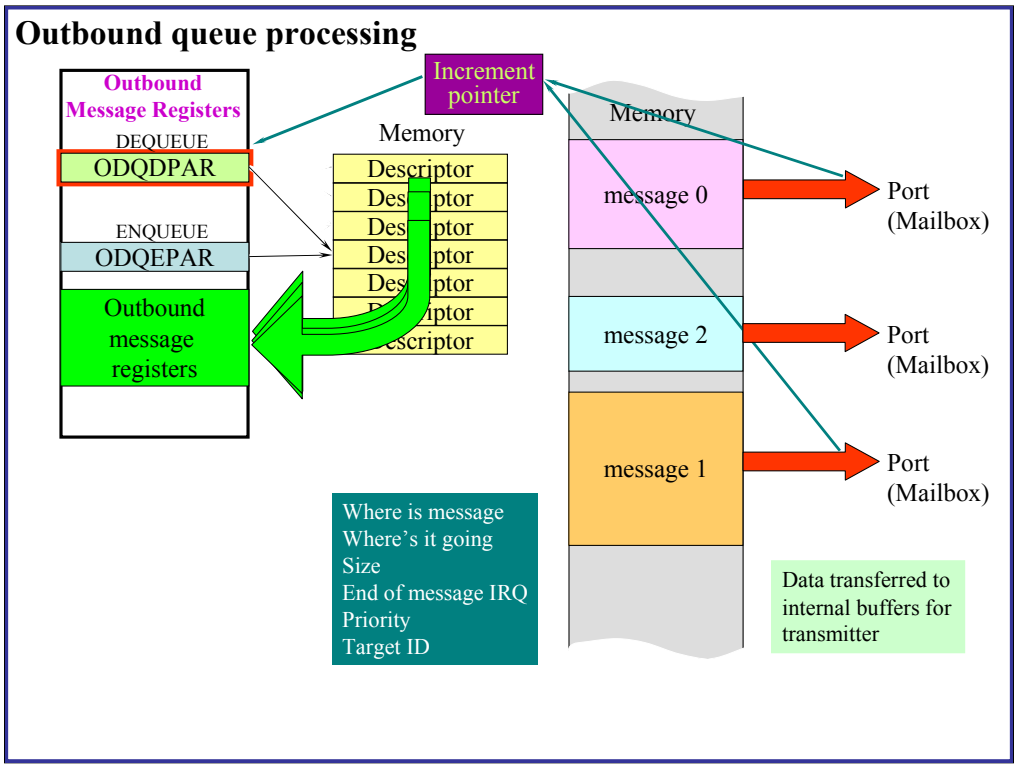
Once the descriptor is completed and the message is ready to be transferred set the MUI in OMR which will cause the controller to increment the ENQUEUE pointer to the next descriptor, and clear the MUI bit.

The process of loading the descriptors can be repeated for as many messages as are needed as long as space is available.

Whenever the ENQUEUE pointer is different from the DEQUEUE pointer the controller fetches the data from the descriptor located where the DEQUEUE pointer indicates, loads it into the control registers for message transmission, and increments the DEQUEUE pointer.

When the DEQUEUE pointer matches the ENQUEUE pointer the controller waits.

As the controller is processing messages it sets the MUB bit in the OSR indicating that it is busy.
When the last segment is sent the busy bit is cleared.



This shows the same structure seen earlier for setting up the queue descriptors, in this case with at least some of the descriptors initialized.

The controller uses the DEQUEUE pointer to access the first descriptor and loads the information from the descriptor into the outbound message registers.

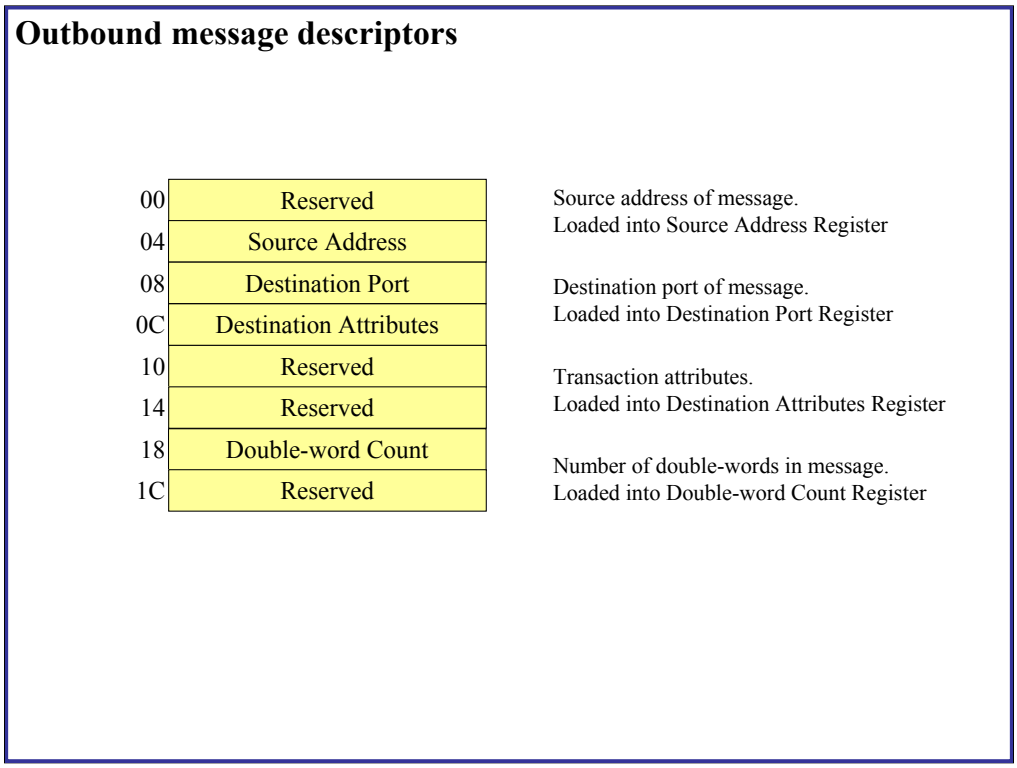
Using that information the controller transfers the message from memory to the internal buffers for the output port.

Once the message is transferred, the controller increments the DEQUEUE pointer to the next descriptor and repeats the process for the second message, and increments the pointer.

If at some stage the DEQUEUE pointer matches the ENQUEUE pointer, which means that the next message is not yet ready, the controller simply waits until the ENQUEUE pointer increments.

As soon as the pointers no longer match the process continues.

This process continues as long as there are messages to send or the end of the queue is reached.



O. k. so what do the descriptors look like !

The descriptors consist of eight thirty-two bit parameters, although only four of them are actually used.

The parameters consist of the source address, destination port, destination attributes, and the double-word count.

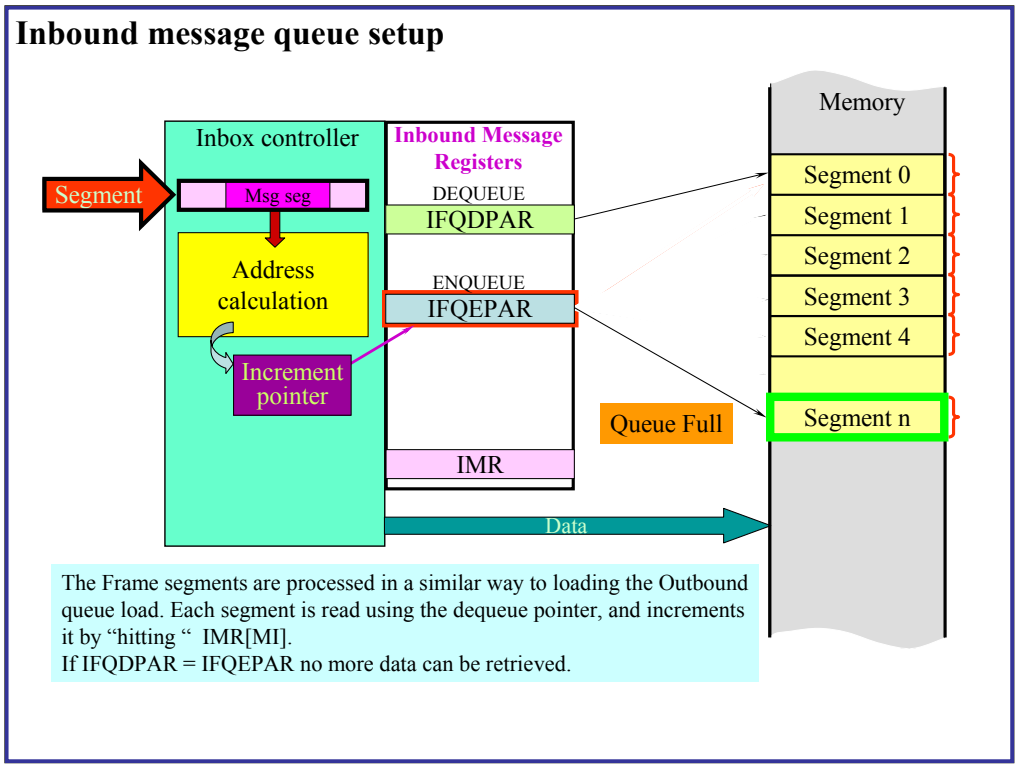
When the controller is processing messages, the source address of the message is loaded into the source address register.

The destination port is loaded into the destination port register.

The transaction attributes are loaded into the Destination attributes register.

And the double-word count is loaded into the double-word count register.

In the latter case be wary, since the name is misleading. The actual value programmed should be the number of bytes of the message.



This animation provides a simple description of the process of receiving inbound messages.

The inbound message registers must be initialized with the inbound ENQUEUE and DEQUEUE pointers both pointing to the first location in memory where the message data is to be stored.

Memory must be allocated to accommodate the expected messages.

When the first segment of a message arrives the header is examined. One field provides the total number of segments for the message, so now the controller knows how many segments must be stored.

Another field in the header indicates the size of the segments of the message, all to be the same, except possibly the last.

Given these two values the controller knows the size of memory needed and most significantly the location in memory where each segment should be stored.

Another field identifies which the current segment is and so the controller stores the data in the correct location, in this case the first, using the ENQUEUE pointer.

When the next segment arrives, the same process occurs, but now the only relevant field for this example is the message segment number.

Now lets look a little closer at what's involved.

Using the information already obtained from the header about the number of segments and segment size the controller can calculate the address for this segment and increment the pointer to the correct location.

The data is then transferred to the appropriate memory location.

If a segment arrives out of order the controller can calculate the correct location for it without any problem, and continue processing the message until it has completed. An indication is provided when the queue is full which can be used, if required to generate an interrupt. One question might be, why would a segment be received out of order? If you remember earlier in the course there was a description of how the packets are transferred through the system. At each stage there is always an acknowledgement, but if a packet cannot be processed at the time, maybe due to a switch being too busy to deal with it, then a retry is returned and that packet is dropped. The sender will transmit that packet again, but that could occur after the next packet had already been sent.

The process continues until all segments are received. When the queue is full an indication is provided by the controller, and can initiate an interrupt.

Once the message is received and the user wishes to read it, a similar process is used to that followed for loading the outbound message queue. In this case no descriptors are used, only the DEQUEUE and ENQUEUE pointers and the inbound mode register. As soon as the queue pointers are not the same the user can read the message from segment zero using the DEQUEUE pointer, and then set the MI bit in the IMR which causes the controller to increment the DEQUEUE pointer by the appropriate value, the segment length. As long as the pointers are different the procedure can continue reading subsequent segments. If at any time the pointers become equal then the next segment cannot be read because it has not yet been received. Once the pointers separate the process can continue until the complete message has been read.

Message Unit Registers - Outbound

OMR Outbound Mode Register

- Descriptor snoop enable
- Circular descriptor queue size - binary count from 2 to 2048 entries
- Queue overflow interrupt enable
- Queue full interrupt enable
- Queue empty interrupt enable
- Message unit transfer mode - chaining or direct
- Message unit increment [MUI]
- Message unit start [MUS]

OSR Outbound Status Register

- Queue full
- Transaction error
- Queue overflow interrupt
- Queue full interrupt
- Message unit busy
- End-of-message interrupt *
- Queue empty interrupt

There are a number of registers that are used to handle the message controller functions and the next three slides indicate the registers for the outbound mailbox and what functions they provide.

The **O**utbound **M**ode **R**egister provides the ability to enable snooping, queue overflow interrupt, queue full interrupt, and queue empty interrupt. It also selects either queue chaining or direct mode, and defines the number of queue descriptors to use.

For controlling the queue operation there is the bit to increment the queue pointer and to start operation.

The **O**utbound **S**tatus **R**egister provides the basic status information, some of which generate the interrupts enabled in the mode register. Interrupt generators are shown in red.

Basic status indicators are queue full, transaction error, and message unit busy. Interrupt generators are overflow, full, empty, and end of message. Note that the end of message interrupt is enabled in the **O**utbound **D**estination **A**ttributes **R**egister.

Message Unit Registers - Outbound continued

ODQDPAR Outbound Descriptor Dequeue Pointer Address Register

- Dequeue pointer address
  bits 0 - 26 of the queue pointer, 32-bit aligned.

Used to indicate the queue entry to use to transfer data out to packets

OSAR Outbound Unit Source Address Register

- Source address of message - most significant 29 bits,
  updated after every transfer.
- Snoop enable

ODPR Outbound Destination Port Register

- Mailbox 0-3 (used for Mbox field in packet)

ODATR Outbound Destination Attributes Register

- End-of-message interrupt enabled *
- Transaction flow level (lowest, next, highest)
- Destination target route (Device ID of Target)

The **O**utbound **D**escriptor **Q**ueue **D**equeue **P**ointer **A**ddress **R**egister provides the twenty-seven most significant bits of the address of the message queue descriptor, used to access the descriptor required to transfer a message segment from the message memory to the outbound message port.

The **O**utbound unit **S**ource **A**ddress **R**egister provides the twenty-nine most significant bits of the source address of the message being sent. This register is automatically updated after every transfer.

The **O**utbound **D**estination **P**ort **R**egister defines the mailbox number of the destination of the packet, and this value is placed into the mailbox field of the packet header.

The **O**utbound **D**estination **A**ttributes **R**egister enables the end of message interrupt, and defines the transaction flow level, the device I.D. of the target which is placed into the associated field of the header.

Message Unit Registers - Outbound continued

ODCR Outbound Double-word Count Register
- Number of double-words in the message
  Binary count from 8 to 4096 bytes

ODQEPAR Outbound Descriptor Enqueue Pointer Address Register
- Enqueue pointer address
  bits 0 - 26 of the queue pointer, 32-bit aligned.

Used to indicate the queue entry to use to load data for messages.

The **O**utbound **D**ouble-word **C**ount **R**egister is actually a byte count of the message to be sent, providing for a minimum byte count of eight (one double word) up to four-thousand and ninety six bytes. Remember all messages must be defined as multiples of double words.

The **O**utbound **D**escriptor **Q**ueue **E**nqueue **P**ointer **A**ddress **R**egister provides the twenty-seven most significant bits of the address of the message queue descriptor used to access the descriptor used to load data into the memory queue for messages.

Message Unit Registers - Inbound

IMR Inbound Mode Register

- Snoop enable
- Frame size - binary count from 8 to 4096 bytes
- Circular descriptor queue size - binary count from 2 to 2048 entries
- Queue full interrupt enable
- Message in queue interrupt enable
- Mailbox increment. Software sets this bit - hardware increments IFQDPAR
- Mailbox enable

ISR Inbound Status Register

- Queue full
- message-in-queue
- Transaction error
- Queue full interrupt
- Mailbox busy (message in progress - cleared on completion or error)
- Message-in-queue interrupt

The next two slides show the registers available for handling the inbound queue.

The **I**nbound **M**ode **R**egister enables the mailbox, snooping, and interrupts for queue full and message in the queue. It also provides the frame size as a byte count, the queue descriptor count, and is used to increment the descriptor pointer when accessing the received message.

The **I**nbound **S**tatus **R**egister provides indications of queue full, message in the queue, transaction error, and mailbox busy, and generates the interrupt for queue full and message in the queue if enabled.

Message Unit Registers - Inbound continued

IFQDPAR Inbound Frame Queue Dequeue Pointer Address Register

- Dequeue pointer address
  bits 0 - 28 of the pointer to the received message in memory.

This is a pointer to the next received message to be process, and is incremented as each message is processed.
It must be initialized to the first inbox message memory location.

IFQEPAR Inbound Frame Queue Enqueue Pointer Address Register

- Enqueue pointer address
  bits 0 - 28 of the pointer to the next memory location to receive a message.

This defines where the next received message is to be loaded into memory. As data is received it is loaded and this pointer incremented. During this operation if this value becomes the same as IFQDPAR then no further data is accepted, and a response of retry is returned .
It must be initialized to the first inbox message memory location.

The **I**nbound **F**rame **Q**ueue **D**equeue **P**ointer **A**ddress **R**egister provides the most significant twenty-nine bits of the address of the descriptor queue used to access the descriptor of the next received message segment. As each segment is accessed this pointer is incremented to the next descriptor.

The **I**nbound **F**rame **Q**ueue **E**nqueue **P**ointer **A**ddress **R**egister provides the most significant twenty-nine bits of the address of the descriptor queue used to store a message segment being received.

Message segments are only stored if this value is different from that of the DEQUEUE pointer.

Output Controller Interrupts

Queue overflow interrupt	ODQEPAR > Queue size queue overflow interrupt bit [QOI] set in OSR if enabled [QOIE] in OMR
Queue full interrupt	ODQEPAR = Queue size queue full interrupt bit set [QFI] in OSR if enabled in [QFIE] in OMR
Queue empty interrupt	ODQDPAR=ODQEPAR queue empty interrupt bit set [QEI] in OSR if enabled [QEIE] in OMR
End of message interrupt	last segment has been transmitted queue empty interrupt bit set [EOMI] in OSR if enabled [EOMIE] in OMR

Queue size defined in OMR[CIRQ_SIZ]

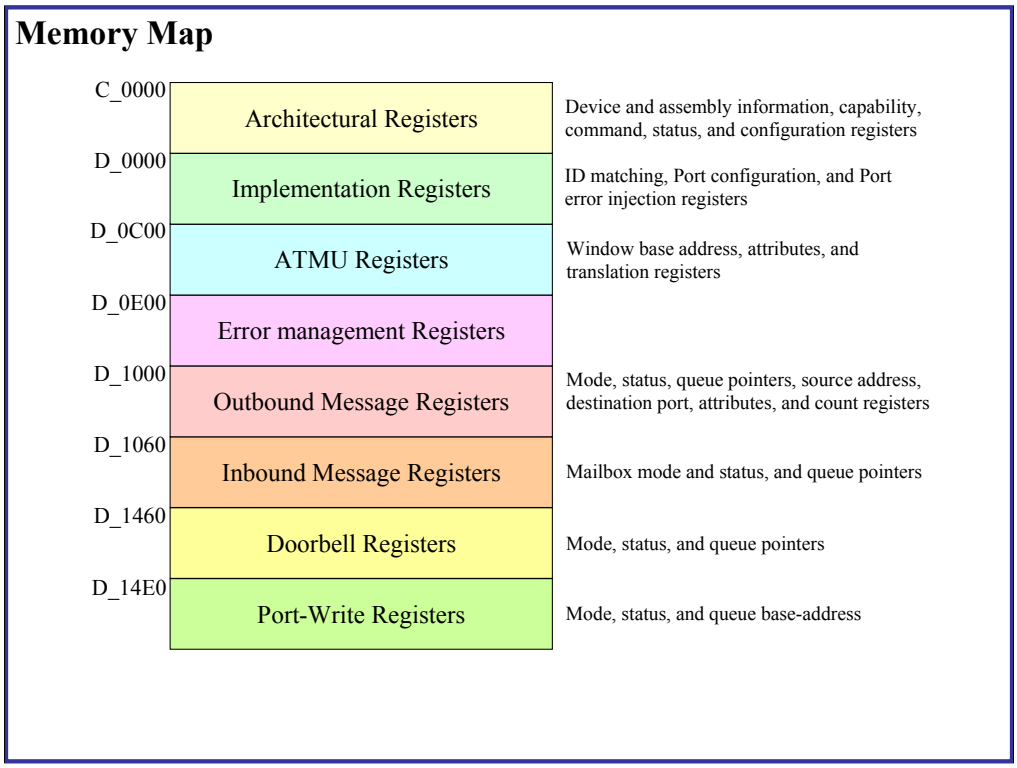
This chart indicates all the interrupts that could be generated for the outbox. They will only be generated if enabled. The queue overflow is generated if the ENQUEUE pointer becomes greater than it should be for the size of queue defined.

The queue full interrupt is generated if the ENQUEUE pointer matches the value indicating that it has reached the queue size defined.

The queue empty is generated if the ENQUEUE pointer matches the DEQUEUE pointer.

The end of message is generated when the last segment of a message is transmitted.

Note that the queue size is defined in the **O**utbound **M**ode **R**egister.



The remaining pages provide a basic indication of all of the registers provided for the RapidIO controller.

They are defined in eight groups shown here. The diagram is not drawn to scale, and some groups are considerably larger than others.

The value shown at the side of each group shows the hexadecimal offset from the top of the RapidIO memory space to where each group starts.

RapidIO Module Registers - Architectural - CAR

Capability Registers - all Read only

Offset	Register	Use
C_0000	DIDCAR	Device and Vendor Identity
C_0004	DICAR	Device Revision
C_0008	AIDCAR	Assembly and Vendor identity
C_000C	AICAR	Assembly revision and extended features
C_0010	PEFCAR	Access availability & support options (DRAM, Local address, mailbox)
C_0014	SPICAR	Port availability & has been read
C_0018	SOCAR	What RIO instructions can be initiated
C_001C	DOCAR	What instructions and functions can be serviced

The architectural group contains the capability and command and status registers,

This diagram shows the capability registers, which are all read only, and indicates the offset to each register and the register designation.

For a complete description of each register see section sixteen-three-two one through eight of the MPC8560 users manual or the equivalent section in the MPC8540 users manual .

RapidIO Module Registers - Architectural - CSR

Command and Status Registers

Offset	Register	Use
C_0040	MSR	R/O Mailbox 0 status (available, full, empty, busy, failed, error)
C_0044	PWDCSR	R/O Doorbell and Write-Port status
C_004C	PELLCCSR	R/O Extended Addressing control (34-bit)
C_005C	LCSBA1CSR	Local reg. config. space base address (14 msb of 34 RIO add)
C_0060	BIDDCSR	Base device ID (Source ID)
C_0068	HBDIDLCSR	Sp Host base device ID that initialized this. (First write only)
C_006C	CTCSR	Component tag ??

Sp = special registers where bits are cleared by writing a '1' to them

The next two diagrams show the command and status registers.

For a complete description of each register see section sixteen-three-two nine through fifteen of the

MPC8560 users manual or the equivalent section in the MPC8540 users manual.

Color coding has been used in the pages to help distinguish some additional features of the registers.

Green indicates registers that are read only.

Blue indicates registers that have some special requirements. The one shown here will only be affected by a write the first time it is written to. Any subsequent writes are ignored.

Others contain bits that are cleared by writing a one to them, a common feature of status bits in the PowerQUICC family. It may also contain some bits which are read only.

Black indicates a standard register with normal read and write bits.

RapidIO Module Registers - Architectural - CSR continued

Command and Status Registers

Offset	Register	Use
C_0100	PMBH0CSR	R/O Extended features pointer and ID
C_0120	PLTOCCSR	R/O Port link Time-out value
C_0124	PRTOCCSR	R/O Port response Time-out value
C_013C	PGCCSR	Agent/Host, enabled, discovery status (Discovery process)[config.pins]
C_0140	PLM**REQ**CSR	Link request command to send
C_0144	PLM**RESP**CSR	R/O Link response - valid, AckID, Link_status
C_0148	PLASCSR	Sp Port Link - next expected AckID, Outstanding unacknowlegedAckID (R/O), Outbound AckID
C_0158	PESCSR	Sp Output errors (retry, cannot forward, stopped, errors, ++)
C_015C	PCCSR	Sp Port Tx. en., Tx. drvr. dis'd., Rx. en., Rx drvr.dis'd., error ch. dis.

This diagram shows the command and status registers.

For a complete description of each register see section 16.3.2.16 through 24 of the MPC8560 users manual or the equivalent section in the MPC8540 users manual.

RapidIO Module Registers - Implementation

Implementation Registers

Offset	Register	Use
D_0000	CR	TID matching - error, accept all, (Match TID or accept all)
D_0010	PCR	Port configuration - Reset time disable, Link time disable, quiesce support, forward before CRC, CRC enable
C_0014	PEIR	Sp Error injection

This diagram shows the implementation registers.

For a complete description of each register see section 16.3.3.1 of the MPC8560 users manual or the equivalent section in the MPC8540 users manual.

RapidIO Module Registers - ATMU Outbound

Address Translation and Mapping Unit

Offset	Register	Use
D_0C00	ROWTAR0	Pointer to RIO Translation address space with 34- bit address & Target ID
		or
		Maintenance transaction Hop count & config. offset
D_0C10	ROWAR0	Attributes - enable, priority, PCI rules, Read type, Write type, outbound window size
D_0C20	ROW**T**AR1	Pointer to RIO Translation address space with 34- bit address & Target ID
		or
		Maintenance transaction Hop count & config. offset
D_0C28	ROW**B**AR1	Base address for window 1 (4 Kbyte block)
D_0C30	ROWAR1	Attributes - enable, priority, PCI rules, Read type, Write type, outbound window size (4KB - 4GB)
D_0C20 - D_0D10		for windows 2 - 8

Note : Window 0 hardwired enabled

This diagram shows the outbound window **A**ddress **T**ranslation and **M**apping **U**nit registers.

With the exception of window zero they are provided as groups of three for the base address, translation address, and attributes for that window.

The base address register defines the local address that is associated with that RapidIO window, and relates to a minimum size of four kilobytes.

The translation address provides most significant bits of the RapidIO thirty-four bit address and the target I.D. supplied for the output transactions.

The attributes register defines outbound transaction attributes such as if the window is enabled, the transaction priority, if it follows PCI rules, the read or write transaction type, and the window size.

There is a set of these registers for eight windows, one through eight, and also the translation and attributes for the default window zero. Any transactions that do not map to one of the other windows automatically is handled by the default, and so no base address is necessary. Affectively this handles transactions which are normally not expected, probably error conditions.

Also be aware that the translation register has an alternative function. If the access matching the window is a maintenance transaction the translation register has a different format, defining hop count and configuration offset.

For a complete description of each register see section 16.3.3.1 of the MPC8560 users manual or the equivalent section in the MPC8540 users manual.

RapidIO Module Registers - ATMU Inbound

Address Translation and Mapping Unit

Offset	Register	Use
D_0D60	RIW**T**AR4	Pointer to INTERNAL Translation address space (32-bit/4KB block)
D_0D68	RIW**B**AR4	Base address for window 4 (RIO 34-bit/4KB block)
D_0C70	RIWAR4	Attributes - enable, target I/F (PCI/local mem), Read type, Write type, Inbound window size (4KB - 4GB)
D_0D80 - D_0D10		for windows 3 - 0

Note : Window 0 hardwired enabled

The inbound address translation and mapping registers perform a similar function to those of the outbound windows except in the opposite direction.

As for the outbound mailbox, window zero is the default, and any incoming packets not mapping to one of the other windows automatically is handled by window zero, so window zero does not have a base address.

The base address contains bits zero through twenty-one of the RapidIO thirty-four bit address.

The attributes register contains the size of the window, the enable bit, the target interface which is either the PCI controller or the local memory, and the type of read or write transaction.

The translation address register provides the local thirty-two bit address that this window maps to.

There are five of each of the attributes and translation registers, zero through four, and four base registers, one through four.

RapidIO Module Registers - Error management

Offset	Register	Use	
D_0E00	PNFEDR	Sp	Fatal errors notification
D_0E04	PNFEDIR		Error notification disable
D_0E08	PNFEIER		Notified error interrupt enable
D_0E0C	PECR	Sp	Indicates valid packet with error notification
D_0E10	PEPCSR0	Sp	First 32-bits of packet with fatal error
D_0E14	PEPR1	Sp	Portion of error packet (type dependent)
D_0E18	PEPR2	Sp	Another portion of error packet (type dependent)
D_0E20	PREDR	Sp	Recoverable error notification
D_0E28	PERTR	Sp	Error recoveries threshold & counter
D_0E2C	PRTR	Sp	Error retries threshold & counter for reporting

This table indicates the error management registers. Using these registers error interrupts can be enabled, and error management service routines can identify the type of error and details necessary to deal with it.

For a complete description of each register see section 16.3.3.3 of the MPC8560 users manual or the equivalent section in the MPC8540 users manual.

RapidIO Module Registers - Doorbell

Offset	Register	Use
D_1460	DMR	Enable, snoop enable, queue size, int enable, increment
D_1464	DSR	status - in Q, in Q int, error, Q full, Q full int, busy,
D_146C	DQDPAR	Queue dequeue pointer- first to process
D_1474	DQEPAR	Queue enqueue pointer- next to add
D_14E0	PWMR	Snoop en, Q full int en, Clear Q, Port-write enable
D_14E4	PWSR	status -Q full, Transaction error, Q full int, Prt-Wr busy
D_14EC	PWQBAR	Port-write base address for data payload.

The doorbell registers provide a similar function to the inbound messages for the inbound doorbell, and enables the controller to deal with an inbound doorbell queue.

For a complete description of each register see section sixteen-three-four-three of the MPC8560 users manual or the equivalent section in the MPC8540 users manual.

That completes this introduction course to the PowerQUICC III RapidIO controller.